ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA
DE TELECOMUNICACIÓN

UNIVERSIDAD DE MÁLAGA





PROYECTO FIN DE CARRERA

INTERCOMUNICACIÓN DE ARDUINOS MEDIANTE CAN-BUS

INGENIERÍA TÉCNICA DE TELECOMUNICACIÓN
SISTEMAS ELECTRÓNICOS

Málaga, 2013 Raúl Milla Pérez

ESCUELA TÉCNICA SUPERIOR DE

INGENIERÍA DE TELECOMUNICACIÓN

UNIVERSIDAD DE MÁLAGA

Titulación: Ingeniería Técnica de Telecomunicación

Especialidad Sistemas Electrónicos

Reunido el tribunal examinador en el día de la fecha, constituido por:

D./D$^{\underline{a}}$. _____

D./D$^{\underline{a}}$. _____

D./D$^{\underline{a}}$. _____

para juzgar el Proyecto Fin de Carrera titulado:

INTERCOMUNICACIÓN DE ARDUINOS MEDIANTE CAN-BUS

del alumno D. Raúl Milla Pérez

dirigido por D. David Santo Orcero

ACORDÓ POR: _____ OTORGAR LA

CALIFICACIÓN DE _____

Y, para que conste, se extiende firmada por los componentes del tribunal, la presente diligencia.

Málaga, a _____ de _____ de 2013

El/La Presidente/a El/La Vocal El/La Secretario/a

Fdo. _____ Fdo. _____ Fdo. _____

ESCUELA TÉCNICA SUPERIOR DE

INGENIERÍA DE TELECOMUNICACIÓN

UNIVERSIDAD DE MÁLAGA

Intercomunicación de Arduinos mediante CAN-Bus

REALIZADO POR:

Raúl Milla Pérez

DIRIGIDO POR:

David Santo Orcero

DEPARTAMENTO DE: Tecnología Electrónica

TITULACIÓN: **Ingeniería Técnica de Telecomunicación**
Sistemas Electrónicos

PALABRAS CLAVE: Arduino, CAN-BUS, SPI, Comunicación, Protocolo, Librería, Hardware, Shield, Industrial, MCP2515.

RESUMEN: Se trata de crear una armadura (Shield), que dote a un Arduino del soporte físico necesario, para comunicarse con otros a través de un bus Controller Area Network (CAN).

Este proyecto no solo persigue el desarrollo del Hardware necesario, sino también la creación de una librería para facilitar su uso. El objetivo es conseguir un producto final con criterios de calidad, eficacia y eficiencia, de fácil construcción y programación y un coste reducido.

Málaga, Junio de 2013

Agradecimientos

Cuando uno empieza la carrera no sabe las complicaciones que puede haber detrás del proyecto más trivial, son los años de experiencia ejerciendo la profesión, la que nos dota de una visión crítica, de forma que nos deja entrever los posibles contratiempos que hay detrás de cada trabajo, proyecto o tarea. Sin duda, el Proyecto Fin de Carrera es el primer reto importante al que, nosotros los ingenieros, nos enfrentamos.

Desgraciadamente, en muchas ocasiones, como en la mía propia, se tiene que compatibilizar con un trabajo remunerado, lo que nos distrae de forma inevitable y hace que la duración de este se prorrogue mucho en el tiempo, hasta llegar incluso, a un punto en el que parece inacabable, inalcanzable... y es gracias a las personas que nos rodean y se interesan por nuestro bienestar, las que nos recuerdan y animan ha terminar lo ya empezado.

Quiero manifestar mi agradecimiento a algunos de quienes, directa o indirectamente, contribuyeron al resultado de este Proyecto Fin de Carrera, ya que me sería imposible mencionarlos a todos. En primer lugar, mis padres, que de forma constante y paciente no han parado de recordarme cual es el camino y me han enseñado a no dejar las cosas a medias. A mi hermano, porque si alguna vez he necesitado ayuda, él siempre ha estado para prestármela.

Sin duda, a mis profesores, en especial a mi tutor de proyecto David Santo, quien hace algunos años me encaminó despertando la chispa en mí sobre el bus CAN, ahora es un protocolo de comunicaciones que me apasiona y espero que algún día mi experiencia con él, me otorgue un puesto de trabajo que me permita seguir experimentando y diseñando, tal como hice en este proyecto.

Recordar de forma especial a Juan Carlos, amigo, socio y compañero de fatigas desde que iniciamos juntos nuestro reto universitario, aún recuerdo aquella mañana invernal el primer día de clases, donde nos conocimos porque ambos llegamos tarde. Desde ese día, muchos son los momentos compartidos, proyectos, asignaturas, desesperaciones y triunfos, sin él, no sería hoy la persona que soy y sin su ayuda y ánimos, este proyecto tampoco.

Por su cariño, amor, perseverancia y buen hacer a mi novia Cristina, porque al igual que mis padres, ha sabido motivarme y apoyarme de forma incondicional para que terminase lo que ya empecé hace bastantes años.

Gracias!

Índice general

Acrónimos **V**

Índice de Figuras **VII**

Índice de Tablas **IX**

1. Introducción **1**

2. Fundamentos **5**

2.1. Introducción a las redes de sensores 5

2.1.1. Topologías de red 6

2.2. Buses de campo 8

2.2.1. Estado del arte 8

2.3. Entornos críticos 9

2.3.1. Ambiente industrial 10

3. Controller Area Network, CAN **11**

3.1. Orígenes del bus CAN 11

3.2. ¿Que es CAN? 12

3.3. Características de redes CAN 14

3.4. El mensaje 16

3.5. Detección y señalización de errores 18

3.6. Un nodo CAN 20
3.7. La capa física de CAN 21
3.7.1. Características eléctricas del bus 21
3.7.2. Cables 22
3.7.3. Terminaciones de línea 22
3.7.4. Longitud del bus vs Velocidad 23
3.7.5. Conectores 23
3.7.6. Temporización y sincronización de bit 23

4. Hardware **25**

4.1. Introducción 25
4.2. Alternativas para un nodo CAN 25
4.3. El CAN-Controller: MCP2515 26
4.4. Controlador y transceiver CAN 27
4.5. Fuente de alimentación 28
4.6. Oscilador 29
4.7. Reset 29
4.8. Buffer de recepción 29
4.9. Sensores externos 30
4.10. Protección por sobrecargas 30
4.11. Mejoras y protección para el Bus 30
4.11.1. Terminación en Split vs Estandard 32
4.11.2. Protección contra el modo común 33
4.11.3. Protección contra transitorios 34
4.12. Prototipos: Diseño, construcción y pruebas 34
4.12.0.1. Esquemático 34
4.12.0.2. Diseño PCB 35
4.12.0.3. Fabricación de prototipos 35
4.12.0.4. Primeras pruebas 37

4.13. Costes y búsqueda de componentes 39

5. Software **43**

5.1. Comunicación SPI 43

5.2. Modos de funcionamiento 48

5.2.1. Configuration Mode 48

5.2.2. Sleep Mode 49

5.2.3. Listen-only Mode 49

5.2.4. Loopback Mode 50

5.2.5. Normal Mode 50

5.3. Máscaras y filtros 50

5.4. Librería para Arduino 51

5.4.1. Control de versiones 52

5.4.2. Empezando con el código fuente 52

5.4.3. Método: write_register 52

5.4.4. Método: read_register 53

5.4.5. Método: begin 54

5.4.6. Método: check_message 55

5.4.7. Método: check_free_buffer 56

5.4.8. Método: get_message 57

5.4.9. Método: send_message 57

5.4.10. Método: SetMode 58

5.4.11. Método: SetRXB0Filters 59

5.4.12. Método: SetRXB1Filters 60

5.4.13. Soporte para interrupciones 61

5.4.13.1. Método: get_message_to_buffer_rx 61

6. Pruebas y Resultados **63**

7. Conclusiones y líneas futuras **67**

Apéndices **70**

A. ArCan **73**

B. Documentos del proyecto **79**

B.1. Manual de Hardware 79

B.2. Manual del Software 79

B.3. Flyer publicitario 80

C. Aplicación Domótica **81**

Bibliografía **85**

Acrónimos

ACK	Acknowledge
API	Application Programming Interface
CAN	Controller Area Network
CD	Collision Detection
CiA	CAN in Automation
CMOS	Complementary Metal Oxide Semiconductor
CNC	Control Numérico Computerizado
CPU	Central Processing Unit
CRC	Cyclic Redundant Check
CSMA	Carrier Sense Multiple Acces
CSMA/CD	Carrier Sense Multiple Acces With Collision Detection
DLC	Data Length Code
ECU	Engine Control Unit
EOF	End Of Frame
GPRS	General Packet Radio Service
GPS	Global Positioning System
HW	Hardware
ICSP	In-Circuit Serial Programming
ID	Identifier
IDE	Integrated Development Environment
IFS	Intermission Frame Space
ISO	International Organization for Standardization
LCD	Liquid-crystal display

MCU	Microcontroller Unit
NRZ	Non-Return to Zero
OSI	Open System Interconnection
PC	Personal Computer
PCB	Printed Circuit Board
PDIP	Plastic Dual-in-Line Package
REC	Receive Error Counter
RTR	Remote Transmission Request
SAE	Society of Automotive Engineers
SMD	Surface Mount Device
SMT	Surface Mount Technology
SOC	System On Chip
SOF	Start of Frame
SPI	Serial Peripheral Interface
SW	Software
TEC	Transmit Error Counter
TVS	Transient Voltage Suppression
USB	Universal Serial Bus

Índice de figuras

1.1. Placa de desarrollo Arduino UNO 1
1.2. Arduino con dos shields, uno GPRS y Ethernet 2

2.1. Topologías de red 6
2.2. Arquitecturas de red 7

3.1. Capas OSI para CAN 13
3.2. Red CAN 14
3.3. Arbitraje en Red CAN con 3 Nodos 15
3.4. Datagrama CAN 17
3.5. Mensaje CAN 18
3.6. Estados de error en un Nodo CAN 20
3.7. Señal diferencial en el Bus 21
3.8. Par trenzado 22
3.9. Temporización de un bit CAN 24
3.10. Forma de Onda 24

4.1. MCP2515 27
4.2. Fuente de alimentación ArCan 28
4.3. Ruido acoplado en el bus con par trenzado 31
4.4. Circuitos de protección para el bus CAN 31
4.5. Terminaciones: Standard y Split 32
4.6. Diagrama de Bode para el fitro de la terminación Split 33

4.7. Diferencia entre terminación estandard y Split 33
4.8. Funcionamiento básico de un TVS 34
4.9. Fotos del 1º prototipo, taladro pasante 35
4.10. Fotos del 2º prototipo, algunos componentes SMD 36
4.11. Fotos del 3º prototipo, fabricación profesional 36
4.12. Fallo en el pin SCK 38
4.13. Pines específicos para shield 39

5.1. SPI Interconexión Maestro-Esclavo 43
5.2. SPI Input Timing 44
5.3. Características del interfaz SPI 45
5.4. Conjunto de Comandos SPI 46
5.5. OpCodes Read 47
5.6. OpCode LoadTX 47
5.7. OpCode RTS 47
5.8. OpCode Status 48
5.9. OpCode RX Status 48
5.10. Máscara y Filtros CAN 51
5.11. Habilitar Interrupción del MCP2515 62

A.1. Esquemático de ArCan Rev C 75
A.2. Gerbers 77

C.1. Diagrama del panel de pruebas 81
C.2. Panel de pruebas 83

Índice de tablas

3.1. Velocidad y distancia en CAN 23

4.1. Tabla de costes de fabricación de la PCB 40
4.2. Tabla de costes para un Shield 41

5.1. Tabla de configuración para el SCK 45
5.2. Dirección de memoria de los buffers de transmisión 58

6.1. Grupos de Pruebas 63
6.2. Plan de Pruebas 65
6.3. Hoja de resultados 66

A.1. Lista de componentes 74
A.2. Nomenclatura en los Gerbers 74

Capítulo 1

Introducción

En estos últimos años, Arduino, un sistema de desarrollo basado en At-Mega8/16, ha propulsado el Hardware libre y ha creado un movimiento, que de no variar el rumbo, será en poco tiempo tan importante como ha conseguido ser el Software libre. Son ya muchos, los que de forma profesional o personal, se dedican a compartir sus avances con el resto de la comunidad, enriqueciendo a todo aquel que de una forma u otra necesite hacer uso de estos conocimientos.

En este proyecto desarrollaremos un "Shield" que conectado a una placa de desarrollo de Arduino como la que vemos en la Figura 1.1 dará soporte a este para comunicarse via CAN-Bus. Cuando hablamos de un "Shield" nos referimos a una placa de circuito impreso que se acoplará a Arduino a fin de incrementar su funcionalidad. En el mercado podemos encontrar multitud de placas de este tipo con un sin fin de funcionalidades, desde un Global Positioning System (GPS), un Liquid-crystal display (LCD) o un controlador para motores de corriente continua. La idea principal que se persigue al crear un "Shield", es ampliar la funcionalidad de Arduino con el objetivo de conseguir un sistema completo y económico, que se ajuste a necesidades concretas.

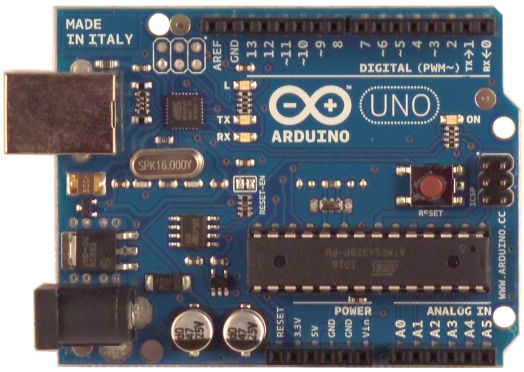


Figura 1.1: Placa de desarrollo Arduino UNO

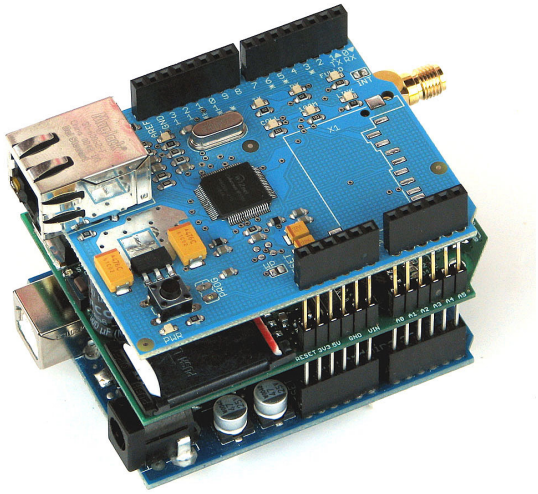


Figura 1.2: Arduino con dos shields, uno GPRS y Ethernet

Por ejemplo, si apilamos un "Shield" General Packet Radio Service (GPRS) y otro GPS podemos conseguir un sistema que actualice la posición en tiempo real de un automóvil por internet, de esta forma, podríamos tener en todo momento la ubicación exacta de un vehículo. Sin duda, la idea de ir apilando funcionalidad según necesidad, es una idea genial para un sistema de desarrollo, que además es económico y goza de un fuerte apoyo en multitud de plataformas de internet.

Para poder entender la importancia que implica el desarrollo de este proyecto, es necesario comprender el impacto que supone usar en el mundo del Hardware libre, uno de los protocolos de comunicaciones más robustos, eficientes y fiables que existen en el mercado actual, CAN, que nace inicialmente como respuesta al incremento de dispositivos electrónicos en el automóvil, aporta una gran cantidad de ventajas en escenarios donde la fiabilidad en las comunicaciones prima frente a otros factores. CAN sigue una topología Cliente-Servidor, que es la más usada en el ámbito industrial; está basado en mensajes, con la potencia que ello implica; cuenta con mecanismos propios de detección de errores, que lo hace muy robusto, utiliza un bus serie, con las ventajas de ahorro de materia prima que conlleva y además es multimaestro, con lo que descarga a la Central Processing Unit (CPU) principal de una gran cantidad de tareas.

Los objetivos iniciales del proyecto consistían únicamente en crear el Shield para Arduino y una librería con una Application Programming Interface (API) que permitiese desarrollar aplicaciones basadas en CAN de forma sencilla. Sin embargo, una vez conocida la ventaja que aporta CAN como protocolo de comunicaciones, los objetivos se han visto claramente incrementados. Introducir el Hardware libre en general y a Arduino en particular en el mundo industrial o acercar las ventajas de CAN a la comunidad son alguna de las nuevas prioridades. Por ejemplo, una posible aplicación para este proyecto, consiste en interconectar hasta 112 Arduinos via CAN-Bus, cada Arduino con sus sensores y de esta forma tendremos una aplicación de sensórica distribuida usando un sistema libre y de bajo coste. Esta gran capacidad de interconexionado hacen de este shield un dispositivo muy capaz en entornos domóticos e industriales. Otros de los posibles usos del proyecto podemos encontrarlo en el mercado de la Automoción, con esta sinergia es posible crear un

sistema libre para diagnosis del automóvil, tan caros en la actualidad, rebajando drásticamente su coste final.

Es evidente que un sistema que dote a Arduino con la capacidad necesaria para comunicarse con redes CAN, abre las puertas del Hardware libre al mundo industrial, beneficiando, sin duda, a gran cantidad de empresas que encuentran trabas económicas a la hora de adquirir equipos CAN y poniendo a disposición de los desarrolladores un conjunto de herramientas para poder empezar a trabajar con este tipo de redes.

Como punto inicial para este proyecto repasaremos el impacto que tiene CAN en el mundo actual. Además, para realizar el desarrollo del proyecto debemos estudiar en profundidad el protocolo a fin de realizar un diseño tanto Hardware como Software correcto y robusto. Una vez que conozcamos el bus, empezaremos el desarrollo del shield, que junto a Arduino constituirán un Nodo CAN. Posteriormente realizaremos la librería para facilitar el control del Nodo CAN para finalizar con un ejemplo práctico y funcional donde usaremos varios Nodos.

Haremos una pequeña descripción más detallada de lo que veremos en los próximos temas:

- **Capítulo 2, Fundamentos**: El objetivo es hacerse una composición de lugar sobre las tecnologías que podemos encontrar en el mercado y las ventajas e inconvenientes de cada una de ellas, para así comprender y entender los beneficios que podría aportar un proyecto de este tipo.
- **Capítulo 3, Controller Area Network, CAN**: Profundizaremos en este tipo de redes, su historia, características tanto físicas como lógicas, tipos de conexionado, ventajas e inconvenientes, estructura de los mensajes, limitaciones...
- **Capítulo 4, Hardware**: Nos centraremos en la construcción del Shield para Arduino, estudiando las alternativas que nos propone el mercado para facilitar el desarrollo, los diferentes dispositivos que podrían mejorar y proteger nuestro circuito, el diseño y la construcción de los primeros prototipos para terminar con una versión estable y lista para la producción.
- **Capítulo 5, Software**: Veremos el proceso de desarrollo de la librería que acompaña al Hardware diseñado, estudiaremos en profundidad el protocolo de comunicaciones SPI y su uso, detallaremos las distintas funciones y métodos que se han implementado para facilitar el manejo del Shield, hablaremos de los registros de configuración y de los distintos comandos que soporta el hardware.
- **Capítulo 6, Pruebas y Resultados**: Propondremos un plan de pruebas con el fin de verificar que tanto Hardware como el Software están correctos y son funcionales, además añadiremos una plantilla como Hoja de resultados.
- **Capítulo 7, Conclusiones y líneas futuras**: Comentaremos que nos ha parecido el proyecto, posibles lineas de desarrollo a futuro y mejoras en las nuevas revisiones, tanto en el Hardware como en el Software.
- **Apéndices**: Encontraremos información para la fabricación, esquemáticos, gerbers, lista de componentes y donde comprarlos, documentación extra, flyer

de publicidad para promocionar el producto, dos pequeños manuales a modo guía de uso para los usuarios finales y una aplicación de prueba haciendo uso de lo desarrollado en este proyecto.

Capítulo 2

Fundamentos

Para poder entender la ventajas que nos aporta CAN en lo que a redes de sensores se refiere, es necesario realizar una pequeña puesta en escena de estos, lo que nos ayudará a tener una visión clara del mercado actual y las posibles aplicaciones según el tipo de protocolo de comunicaciones que usemos. Además, veremos a grandes rasgos las topologías de red y la arquitectura e interconexión de nodos más comunes, centrándonos en los Buses de Campo y comentando ventajas y desventajas en cada caso, finalizando con el estado del arte en la actualidad.

2.1. Introducción a las redes de sensores

Se conoce como tal, al conjunto de sensores y/o actuadores, equipados con una pequeña unidad de proceso, capaz de gestionar las comunicaciones entre ellos a la vez que realiza las distintas tareas de medición/acción. Cada elemento independiente de la red es conocido como "nodo" y debe ser totalmente autónomo.

En la actualidad y gracias a los grandes avances científicos en la miniaturización de los componentes electrónicos y su bajo coste, se han conseguido crear nodos muy pequeños, económicos, inteligentes, dotados de comunicaciones inalámbricas y con un consumo eléctrico muy reducido, esto ha propiciado lo que hoy es conocido como polvo inteligente o "smart dust".

Las smart city, los sistemas de telemonitorización, el entorno inteligente y otras muchas aplicaciones relacionadas con la sensórica distribuida, se están conviertiendo en un campo ampliamente demandado y donde los avances nos sorprenden a diario. Una gran parte de las redes de sensores se intercomunica de forma inalámbrica, aunque cuando entramos en entornos más exigentes y críticos, como los industriales, donde el ruido de la maquinaria, los metales, etc, pueden afectar de forma negativa y tajante a la mayoría de los sistemas de comunicación vía radio, vemos que los más usados como Zigbee, Z-wave, Wifi, Bluetooth, se convierten en sistemas ineficientes y claramente peligrosos.

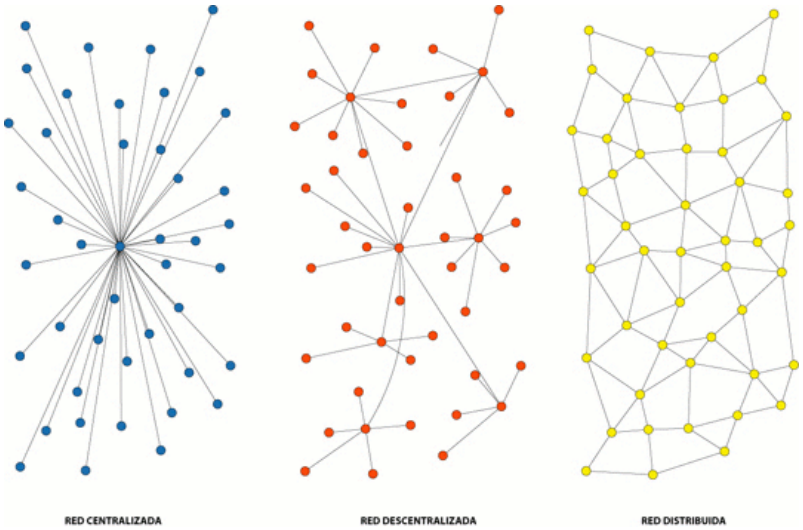


Figura 2.1: Topologías de red

Imagínense un robot scorbo de 5 metros que monta piezas en una planta de ensamblaje de automóviles, donde todo el perímetro está dotado de gran cantidad de sensores que ante cualquier evento "extraño", como podría ser la intervención de personal humano, se comunicase mediante sistemas inalámbricos. Si por cualquier motivo, el sensor que detecta la presencia humana no consigue comunicar a la unidad de control el evento, debido a alguna interferencia, porque pierda el paquete de datos, o se retrase, el obrero podría sufrir un accidente, probablemente grave.

Vemos claramente como en entornos complicados, la mayoría de las tecnologías usadas actualmente para la sensórica de "consumo", es ineficiente y claramente peligrosa. En estos casos, es mejor optar por sistemas de comunicaciones robustos y tolerantes a fallos e interferencias. Actualmente en el mercado todas las opciones con estas características pasan obligatoriamente por soluciones cableadas.

2.1.1. Topologías de red

Se conoce como las distintas formas de interconexión entre nodos. Podemos agruparlas en tres categorías tal y como vemos en la Figura 2.1:

- Redes centralizadas: Todos los nodos, menos uno, son periféricos y sólo pueden comunicarse a través del nodo central. La caída del nodo central priva del flujo de información a todos los nodos.

- Redes descentralizadas: Es producto de la interconexión de los nodos centrales de varias redes centralizadas. La caída de uno de los nodos centralizadores, conlleva la desconexión de uno o más nodos del conjunto de la red.

- Redes distribuidas: Todos los nodos se conectan entre sí, sin que tengan que pasar necesariamente por uno o varios centros. La red es robusta ante caída de nodos, puesto que ningún nodo al ser extraido genera la desconexión de otro.

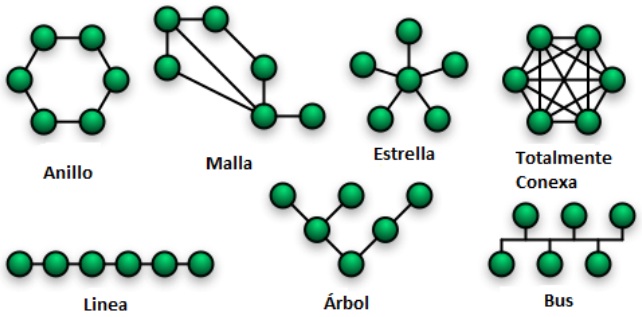


Figura 2.2: Arquitecturas de red

En necesario matizar dos aspectos diferentes a la hora de considerar una topología:

- Topología física o también conocida como "Arquitectura de red", es la disposición real de los nodos y el cableado.
- Topología lógica, es la forma en que los nodos se comunican a través del medio físico.

Por tanto, vemos que no necesariamente deben coincidir ambas topologías. En la Figura 2.2 podemos observar algunas de las posibles formas de interconexión física.

- Anillo: Los nodos están unidos unos con otros formando un círculo por medio de un cable común. El último nodo de la cadena se conecta al primero cerrando el anillo. Las señales circulan en un solo sentido alrededor del círculo, regenerándose en cada nodo. Con esta metodología, cada nodo examina la información que es enviada a través del anillo. Si la información no está dirigida al nodo que la examina, la pasa al siguiente en el anillo. La desventaja del anillo es que si se rompe una conexión, se cae la red completa.
- Estrella: La red se une en un único punto, nodo central, y este suele ser el encargado de gestionar toda la red. Este esquema tiene una ventaja al tener un panel de control que monitorea el tráfico y evita las colisiones y una conexión interrumpida no afecta al resto de la red. La principal ventaja es que la caída de un nodo periférico no afecta a la red, sin embargo, si se trata del central, la red cae al completo.
- Bus: Permite que todas las estaciones reciban la información que se transmite, una estación transmite y todas las restantes escuchan. Consiste en un cable con un terminador en cada extremo del que se cuelgan todos los elementos de una red. Todos los nodos de la red están unidos a este cable: el cual recibe el nombre de "Cable Backbone". Sus principales ventajas son, económico, fácil de montar y la caída de un nodo no produce la de la red. Su problema es obvio, si el Backbone se avería la red cae al completo.
- Híbridas, arquitecturas como la de Bus, Estrella y Anillo se combinan algunas veces para formar redes híbridas, tales como árbol, malla...

Añadir cable en grandes instalaciones, repercute mucho en los gastos de instalación y es un criterio importante a la hora de buscar posibles soluciones. En los entornos donde encontramos problemas con las redes inalámbricas, con un entramado bastante denso de sensores/actuadores y que requieran características de redes distribuidas, será interesante optar por un conexionado tipo bus, porque la información se transmite a todos los nodos y es cada uno el que decide que hacer con ella, por otra parte, el coste en cableado es mucho menor que en las demás arquitecturas.

2.2. Buses de campo

Es un término genérico que describe un conjunto de redes de comunicación para uso industrial, cuyo objetivo inicial es sustituir las conexiones punto a punto, entre los elementos de campo y el equipo de control, a través del tradicional bucle de tensión o de corriente 4-20 mA.

Cada dispositivo de campo, incorpora cierta capacidad de proceso que lo convierte en un dispositivo inteligente, manteniendo siempre un bajo coste. Cada uno de estos elementos, será capaz de ejecutar funciones simples de auto-diagnóstico, control o mantenimiento, así como comunicarse bi-direccionalmente a través del bus. Además estos dispositivos pueden comunicarse tanto con el nodo central de control, como con los demás dispositivos, con todas las ventajas que ello implica.

El objetivo es reemplazar los sistemas de control centralizados, por redes distribuidas, con el fin de mejorar la calidad, la eficiencia y reducir los costes.

La ventaja principal de los buses de campo que lo hacen tan atractivos a los usuarios finales, es la reducción de costes, debido a tres causas, ahorro en la instalación, en el mantenimiento y aquellos derivados de la mejora del rendimiento.

Además, ofrecen mayor flexibilidad al usuario en el diseño de sistemas. Algunos algoritmos y procedimientos de control que con sistemas de comunicación tradicionales debían incluirse en los propios algoritmos de control, radican ahora en los dispositivos de campo, simplificando el sistema de control y sus posibles ampliaciones.

Otra de las ventajas que tienen los buses de campo, es que suelen reducir a tres capas el modelo Open System Interconnection (OSI), física, enlace y aplicación, para hacer más ágil su funcionamiento.

Si nos centramos en el problema de las redes de comunicaciones industriales, debemos resaltar el concepto de los “buses de campo”. Actualmente existen varias capas de aplicación, pero las mayoría de ellas se soporta sobre RS485.

2.2.1. Estado del arte

- ASI “Actuator Sensor Interface”, es el bus más inmediato en el nivel de campo y más sencillo de controlar. Consiste en un bus cliente/servidor con un máximo

de 31 participantes que transmite por paquetes de solo 4 bits de datos. Es muy veloz, con un ciclo de 5 ms aproximadamente. Alcanza distancias de 100 m o hasta 300 m con ayuda de repetidores.

- BITBUS, es el más difundido en todo el mundo; es del tipo cliente/servidor y admite como máximo 56 clientes; el paquete puede transmitir hasta 43 bytes de datos.
- Profibus, "PROcess FIeld BUS", es el estándar europeo en tecnología de buses; se encuentra jerárquicamente por encima de ASI y BITBUS, trabaja según procedimiento híbrido token passing, y dispone de 31 participantes hasta un máximo de 127. Su paquete puede transmitir un máximo de 246 bytes, y el ciclo para 31 participantes es de aproximadamente 90 ms. Alcanza una distancia de hasta 22300 m.
- FieldBus se basa en un subconjunto de tres capas de la arquitectura OSI. Fieldbus ocupa los niveles 1 (Físico), 2 (Enlace de Datos) y 7 (Aplicación); teniendo en cuenta que este último no solo se encarga de la interfaz de usuario sino de aplicaciones específicas, dependiendo de cada aplicación

Por tanto, algunos atributos que se consideran necesarios en los buses de campo podrían ser:

- Reducción masiva de cables.
- Distancias grandes de conexionado.
- Dispositivos inteligentes.
- Control distribuido.
- Baja vulnerabilidad a interferencias.
- Mecanismos de detección y correción de errores.
- Alta fiabilidad.
- Prioridad de mensajes.

2.3. Entornos críticos

Sin duda, la gestión de los frenos de un automóvil o el control de los motores de un avión, son entornos claramente críticos en los que un fallo puede producir un accidente grave, en donde la vida se ponga en riesgo, es por ello, que podemos encontrar en el mercado protocolos de comunicación muy robustos y adecuados a las exigencias económicas que impone el mercado. Sería una locura cablear un avión, por peso y coste, usando otras arquitecturas distinta a la de bus. Actualmente, para los entornos descritos se usa CAN', que cumple con todos los requisitos necesarios, tanto en velocidad como en seguridad, usando además, una arquitectura física económica tipo Bus.

2.3.1. Ambiente industrial

De igual forma que en los casos anteriores, los entornos industriales son muy ruidosos y propicios a problemas de comunicaciones, por ello se descarta también usar tecnologías inalámbricas. Cuando hicimos el análisis del estado del arte, exponía algunos de los buses de campos que se usan en entornos industriales, pero actualmente el CAN está cogiendo fuerza por sus excelentes características y cada vez es más común encontrar maquinaria industrial dotada con esta interfaz de comunicaciones.

Capítulo 3

Controller Area Network, CAN

3.1. Orígenes del bus CAN

El sistema de bus serie CAN, siglas cuyo significado en castellano es "Control de Área de Red", nació en febrero de 1986, cuando el grupo "Robert Bosch GmbH", más conocido como "Bosch" lo presentó en el congreso de la Society of Automotive Engineers (SAE). Desde entonces, CAN se ha convertido en uno de los protocolos líderes utilizando un bus serie.

A comienzos de los 80, los ingenieros de Bosch evaluaron el posible uso de los sistemas de bus serie existentes en los coches de pasajeros, pero ninguno de los protocolos de red disponibles satisfacían los requisitos de estos, debidos generalmente por fiabilidad y robustez. El ingeniero Uwe Kiencke fue quien inició el desarrollo del nuevo sistema de comunicaciones en 1983.

Este protocolo de bus serie se ideó principalmente para aportar mayor funcionalidad, seguridad y fiabilidad, junto a una mayor eficiencia en el gasto del combustible, ya que la reducción del peso y la complejidad de los automóviles, a través de la reducción del cableado, iban a favorecer este hecho.

Ingenieros de "Mercedes-Benz" pronto se involucraron en las primeras fases de creación del nuevo sistema junto a "Intel" como potencial vendedor de semiconductores. El profesor Wolfhard Lawrenz de la Universidad de Ciencias Aplicadas de Brunswick-Wolfenbüttel, Alemania, fue quién dio nombre de "Controller Area Network" al nuevo protocolo de red.

A mediados de 1987, "Intel" presentó el primer chip controlador CAN, el 82526. Poco tiempo después "Philips Semiconductors" sacaría al mercado 82C200.

Estos dos antepasados de los controladores CAN actuales, eran completamente distintos en cuanto a filtros de aceptación y control de mensajes: Intel adoptó el concepto de FullCAN que requería menos carga de la CPU del microcontrolador a diferencia de la implementación BasicCAN elegida por Philips. Por otra parte,

el dispositivo "FullCAN" era limitado con respecto al número de los mensajes que podía recibir. Además, el controlador "BasicCAN" requería menos silicio, lo que abarataba más su coste. Actualmente, los términos "BasicCAN" y "FullCAN" han quedado obsoletos.

Esta primera especificación del bus fue mejorada y ampliada (CAN 2.0) por Bosch en 1991, lo que permitió que empezara a utilizarse masivamente en la industria automovilística europea. Mercedes-Benz fue el primer fabricante de coches en incorporar CAN, y junto a otros grandes fabricantes europeos de automóviles, fundaron el grupo CAN in Automation (CiA) (Can in Automotion), creadora de la capa de aplicación para el bus. Aunque varias empresas crearon sus especificaciones para las redes CAN, La International Organization for Standardization (ISO) publicó en 1993 un estándar, el ISO11898, que fue ampliado en 1995 junto con la aparición del protocolo unificado de la CiA para el CAN, el CANopen.

A pesar de que CAN surgió con la idea de ser utilizado en los automóviles de pasajeros, un gran número de las primeras aplicaciones llegarían desde otros segmentos de mercado distintos. Entre otros, el fabricante finlandés de ascensores "Kone", fue uno de los primeros en aprovecharse de las ventajas que aportaba el nuevo bus. Por ejemplo, la oficina de ingeniería Suiza "Kyaser" lo introdujo a los fabricantes de maquinaria textil del país, que acabaron fundando el "Grupo de usuarios textiles CAN", bajo el liderazgo de "Lars-Berno Fredriksson".

Otro caso se produjo en Holanda, "Sistemas médicos Philips" lo utilizó para las redes internas de sus máquinas de Rayos X. La "especificación para mensajes de Philips" (PMS), desarrollada principalmente por Tom Suters, representó la primera capa de aplicación para redes CAN. También se sucedieron todo tipo de propuestas académicas, como por ejemplo, la creación a finales de los 80 de un sistema basado en CAN para vehículos agrícolas (LBS).

En el año 2000 se define y desarrolla el protocolo de comunicaciones en tiempo real para CAN, denominado TTCAN (Time-Triggered Communication of CAN), bajo el estándar ISO11898-4.

3.2. ¿Que es CAN?

CAN es un protocolo de comunicaciones basado en una topología bus para la transmisión de mensajes en ambientes distribuidos, además ofrece una solución a la gestión de la comunicación entre múltiples unidades centrales de proceso. Se diseñó principalmente para el sector del automóvil como una respuesta a comunicar la creciente cantidad de elementos introducidos en los automóviles actuales, que incluyen los elevalunas, la centralita de confort, la alarma, etc. Al utilizar un bus para la comunicación, se evita tener que introducir cableado para el paso de tantas señales entre elementos, ahorrándose así una elevada cantidad de cobre y por consiguiente una reducción de los costes y peso, lo que desembocaría de forma indirecta en el consumo de combustible tal y como mencionamos en el apartado anterior.

CAN proporciona beneficios al ser un protocolo de comunicaciones normalizado,

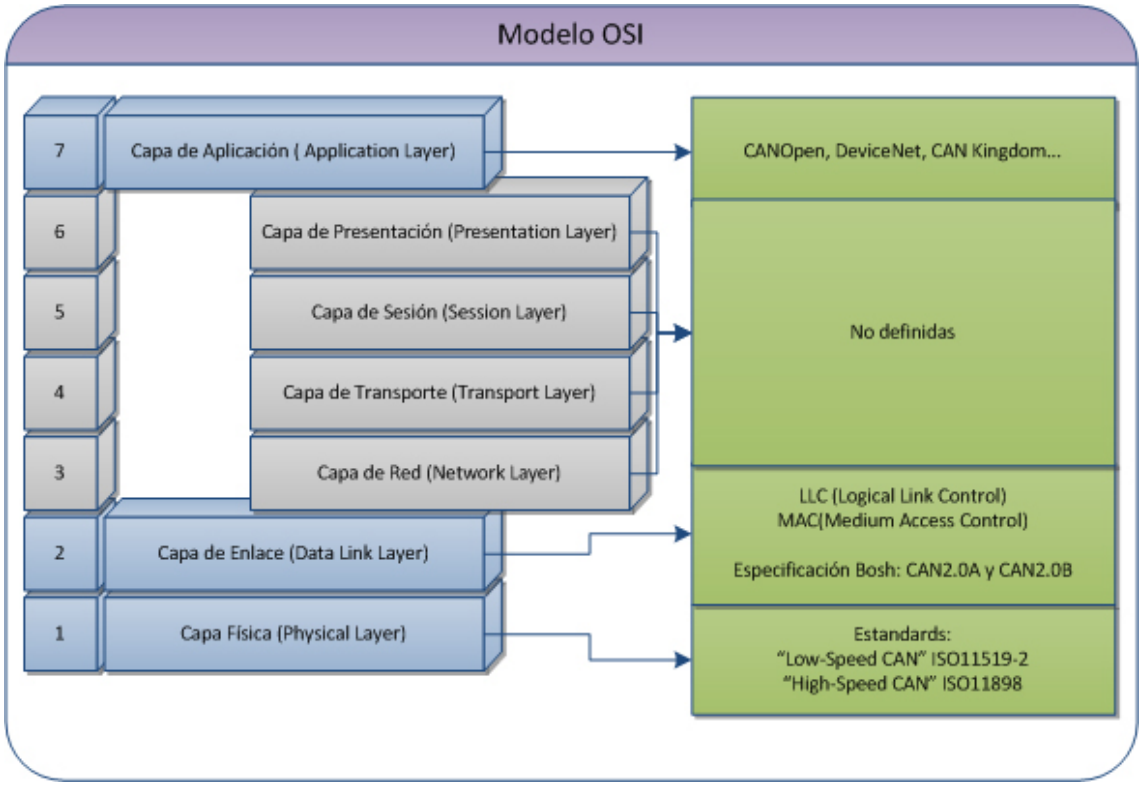


Figura 3.1: Capas OSI para CAN

con lo que se simplifica y economiza la tarea de comunicar subsistemas de diferentes fabricantes sobre una red común o bus. Otra gran ventaja es que el procesador anfitrión (host) delega la carga de comunicaciones a un periférico inteligente, por tanto el procesador dispone de mayor tiempo para ejecutar sus tareas.

Con referente a la estandarización de las redes CAN , es necesario saber que únicamente está definido las dos primeras capas del modelo OSI, "Fisical Layer" y "Data Link Layer", dejando libre a los fabricantes la implementación de la última capa, "Application Layer". Para concretar vamos a describir la funcionalidad de las dos capas estandards:

- "Fisical Layer" o capa física: Se encarga de las conexiones físicas de la red, especifica los niveles o características eléctricas de las señales así como la codificación, decodificación, temporización y sincronización de los bits del frame.

- "Data Link Layer" o capa de enlace de datos: Controla el flujo de información entre los nodos de la red. Es decir, se encarga de la transmisión de los bits en frames de información, se ocupa de que los mensajes lleguen a su destino sin errores, controla las secuencias de transmisión, los acuses de recibido y si en determinado caso no se recibió bien un mensaje, se encarga de retransmitirlo.

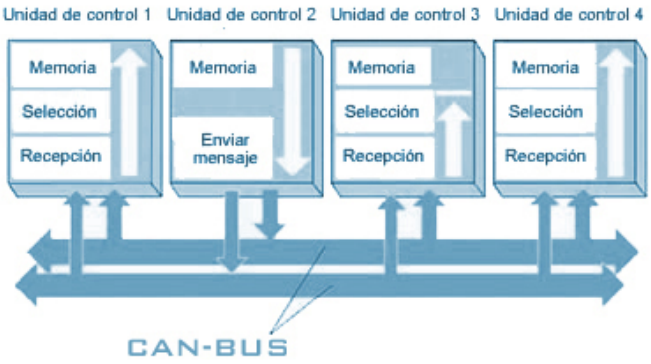


Figura 3.2: Red CAN

3.3. Características de redes CAN

CAN es un protocolo serie asíncrono del tipo Carrier Sense Multiple Acces With Collision Detection (CSMA/CD). Al ser Carrier Sense Multiple Acces (CSMA), cada nodo de la red debe monitorizar el bus y si detecta que no hay actividad, puede enviar un mensaje. Al ser Collision Detection (CD), si dos nodos de la red comienzan a transmitir a la vez un mensaje, ambos detectan la colisión, un método arbitrario basado en prioridades resuelve el conflicto. El bus usa un medio compartido mediante multiplexación. Se trata de un protocolo "Multicast", es decir, todo el mundo puede transmitir, de uno en uno, y escuchar. Por ejemplo, un nodo envía un mensaje a través del bus y este lo reciben todos los componentes de la red que se encargan de saber si la información del mensaje le es útil o no. Si el mensaje fuese de interés para algún nodo, este lo almacena y procesa, si no, simplemente la deshecha.

En escenario que podemos ver en la Figura 3.2, la "Unidad de control 2", envía el mensaje que tenía almacenado en memoria al bus, y todas las demás unidades ven ese mensaje a sus entradas. Sin embargo la "Unidad de control 3" deshecha este mensaje en la etapa de "Selección", mientras que las restantes deciden que es apropiado y lo almacenan.

CAN está orientado a mensajes, es decir la información que se va a intercambiar, se descompone en mensajes, a los cuales se les asigna un identificador (ID) y entre 0 y 8 bytes de datos. El campo ID de cada trama no identifica al nodo que envía el mensaje sino al tipo de mensaje. Es responsabilidad del diseñador de cada aplicación definir como se asignan los ID a las tramas generadas por los diferentes nodos del sistema. Este ID o identificador es el que consigue que el nodo pueda saber, si el mensaje le será útil o no.

Existen dos formatos posibles para el encabezado de las tramas, que difieren en el campo ID y en algunos campos auxiliares de control. En el formato estándar el campo ID es de 11 bits. Este formato era el único disponible en la primera versión del protocolo. En el formato extendido en cambio, el campo ID está formado por la concatenación del ID básico de 11 bits con el ID extendido de 18 bits, totalizando 29bits.

CAN usa el concepto de "Bit Dominante" y "Bit Recesivo", lo que quiere decir

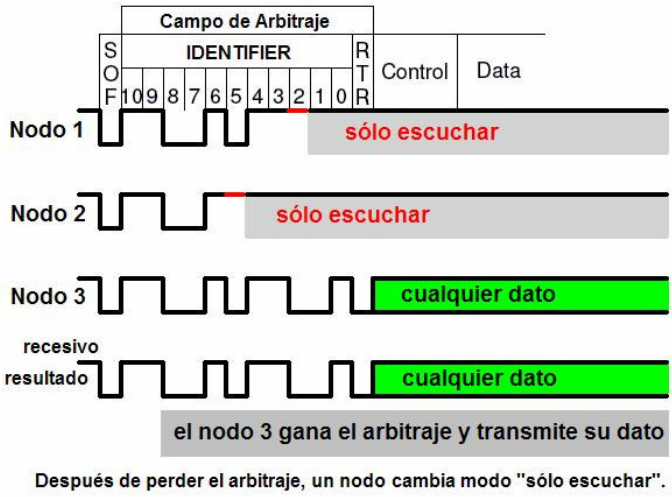


Figura 3.3: Arbitraje en Red CAN con 3 Nodos

que el bit dominante prevalece siempre sobre el bit recesivo. En este caso, el '0' lógico es el dominante y por tanto siempre tendrá prioridad sobre el '1'. Haciendo uso de este concepto, se entiende que hay un sistema de prioridades en los mensajes implementado en la propia capa física.

Si se produce una colisión entre dos nodos que inician la transmisión de una trama al mismo tiempo, se utiliza un mecanismo de arbitraje por prioridad para resolver el conflicto. El sistema compara cada bit a medida que los contendientes escriben sobre el bus el campo ID y algunos bits adicionales del encabezado de la trama (IDE que indica el formato de ID y RTR que indica el tipo de trama), que conjuntamente se denominan campo de arbitraje. El nodo que escribió un valor recesivo lo detecta porque lee del bus un valor dominante y debe retirarse de la contienda convirtiéndose en receptor. El nodo que puso el valor dominante sigue adelante con la transmisión sin notar siquiera que hubo un conflicto. El nodo perdedor podrá reintentar la comunicación una vez finalizada la transmisión de la trama por el ganador. En la Figura 3.3 vemos un ejemplo de arbitraje en una red CAN con 3 nodos.

El protocolo CAN fue desarrollado para ser utilizado en sistemas con fuertes requerimientos de fiabilidad, por lo que posee múltiples mecanismos de detección de errores: error de bit, error de bit de relleno (stuff), error de CRC, error de trama, error de ACK.

Como resumen y de forma rápida las principales características de una red CAN son:

- Prioridad de mensajes.
- Bus serie con dos hilos.
- Garantía de tiempos de latencia.
- Flexibilidad en la configuración.
- Recepción por multidifusión (multicast) con sincronización de tiempos.
- Sistema robusto en cuanto a consistencia de datos.

- Sistema multimaestro.
- Detección y señalización de errores.
- Retransmisión automática de tramas erróneas
- Distinción entre errores temporales y permanentes de los nodos de la red, y desconexión autónoma de nodos defectuosos.

Es muy interesante saber que la ISO define dos tipos de redes CAN:

- Red de alta velocidad: Capaz de alcanzar 1 Mbps, bajo el estándar ISO11898-2, destinada a controlar el motor e interconectar las Engine Control Unit (ECU).
- Red de baja velocidad: Tolerante a fallos que permite alcanzar 125 Kbps, bajo el estándar ISO11519-2/ISO11898-3, dedicada a la comunicación de los dispositivos electrónicos internos de un automóvil como son el control de puertas, techo solar, luces, asientos, etc.

3.4. El mensaje

Para entender mejor como funcionan las redes CAN, es necesario comprender la estructura que componen los mensajes que se envían a través del bus. Existen dos tipos de mensajes CAN que se distinguen únicamente por la longitud del Identificador "Identifier". En el caso del Formato Estandar "Standard Message Format" son 11 bits, mientras que para el Formato Extendido "Extended Message Format" son 29 bits.

En las redes CAN, no se asigna a los dispositivos una dirección ni tampoco ningún mecanismo que los difiera entre ellos, es una capa superior software, la capa "Selección", la que se encarga de saber si el mensaje le concierne o no, y lo sabe gracias al Identificador. Esto es una característica tan curiosa como potente y es que en una red CAN un mismo mensaje puede ser recibido por varios dispositivos, reaccionando cada uno de forma particular ante una misma trama.

En la Figura 3.4 podemos ver los distintos campos de los que se componen un mensaje CAN.

Veamos en mayor detalle el significado y el fin de cada campo:

- Start of Frame (SOF): Este bit se encarga de avisar a los demás dispositivos que se va a iniciar un mensaje, y de esta forma se sincronizan. El inicio de mensaje se marca por un bit dominante '0'.
- "Arbitration Field": Consta del identificador del mensaje, 11 bits, y un bit de control adicional llamado Remote Transmission Request (RTR). Cuanto más bajo sea el valor del Identificador más prioridad tendrá el mensaje, debido al concepto de bit dominante y recesivo. Durante la transmisión de este campo,

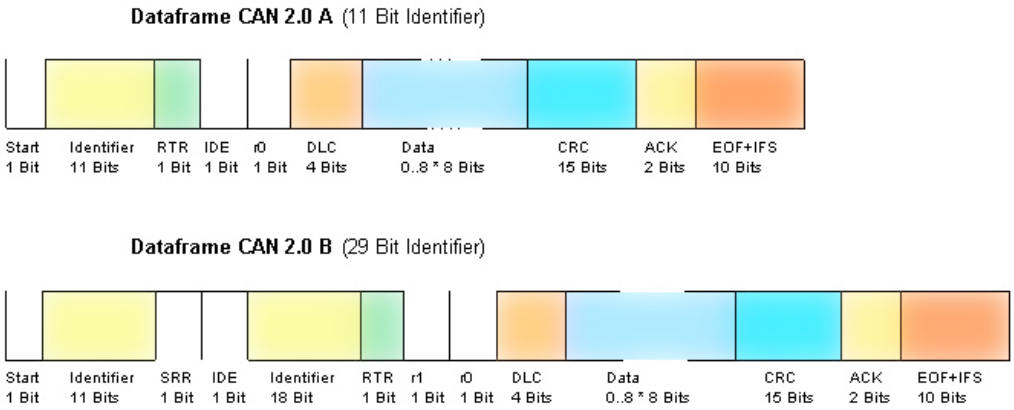


Figura 3.4: Datagrama CAN

el emisor comprueba en cada bit si todavía está autorizado para emitir o si esta emitiendo otro dispositivo con un mensaje de mayor prioridad. El bit RTR indica si el mensaje contiene datos (RTR=0) o si se trata de una trama remota sin datos (RTR=1). Una trama de datos siempre tiene una prioridad más alta que una trama remota. La trama remota se emplea para solicitar datos a otras unidades o bien porque se necesita para realizar un chequeo.

- "Control Field": Este campo informa sobre las características del "Data Field", se compone por un primer bit IDE, que indica que tipo de mensaje es, '0' para una trama estándar y '1' para una trama extendida. Después un bit reservado y los cuatro últimos, Data Length Code (DLC), contienen la longitud en Bytes del campo de datos "Data Field".

- "Data Field": En este campo se encuentra la información que deseamos transmitir, puede variar entre 0 y 8 Bytes. Un mensaje de longitud 0 puede emplearse para la sincronización de procesos distribuidos.

- Cyclic Redundant Check (CRC): Garantiza la integridad del mensaje con un código de 15 bits para verificar posibles errores de transmisión, está basado en una codificación Hamming con distancia 6, el último bit es siempre un '1' y delimita el campo CRC.

- Acknowledge (ACK): Está compuesto por dos bit que son siempre trasmitidos como recesivos '1'. Todas los dispositivos que verifican el CRC modifican el primer bit del campo ACK por uno dominante '0', de forma que el periférico que está todavía trasmitiendo reconoce que al menos algún dispositivo ha recibido el mensaje correctamente. De no ser así, el emisor interpreta que su mensaje presenta algún error.

- End Of Frame (EOF): Indica el final del mensaje con una cadena de '7 bits recesivos '1'. Puede ocurrir que en determinados mensajes se produzcan largas cadenas de '0' y '1' o unos, y que esto provoque una pérdida de sincronización entre los dispositivos. CAN resuelve esta situación insertando un bit de diferente polaridad cada cinco bits iguales: cada cinco '0' se inserta un '1' y viceversa. El dispositivo que utiliza el mensaje, descarta un bit posterior a cinco bits iguales. Estos bits reciben el nombre de "Bit Stuffing" o bit de relleno.

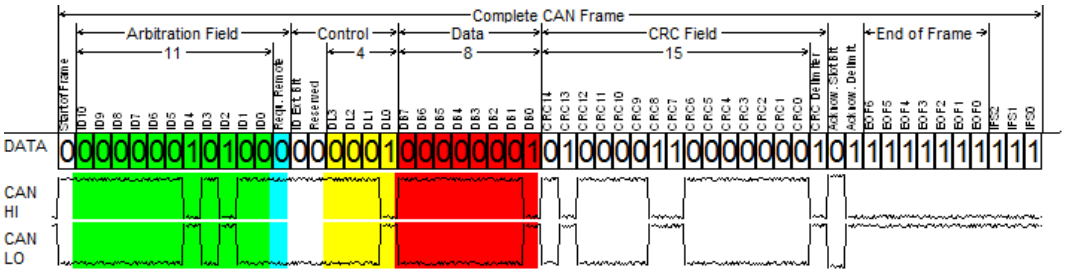


Figura 3.5: Mensaje CAN

- Intermission Frame Space (IFS): Es el espacio mínimo de tiempo entre mensajes y consta 3 bits recesivos.

En la Figura 3.5 podemos observar la equivalencia física de los '0' y los '1' de un mensaje CAN transmitido a través del bus.

3.5. Detección y señalización de errores

En el protocolo CAN se ha realizado un importante sistema de manejo de errores. Este sistema permite detectar errores en los mensajes que se transmiten para así, en el caso que sea necesario, se re-transmitan los erróneos. Si un controlador del bus detecta un error, enviará un flag de error para avisar del mismo y así destruir el tráfico del bus. Los otros nodos posteriormente, detectarán un error debido a la violación de la regla del bit stuffing en el flag de error y enviarán otros flags de error.

El protocolo CAN define cinco tipos de detección de errores, de los cuales dos se realizan a nivel de bit en la capa física (OSI layer 1) y el resto es a nivel de trama en capa de enlace de datos o (OSI layer 2):

- "Monitoring" (OSI Layer 1): Cuando un nodo está transmitiendo, éste monitoriza el nivel del bus y si el bit que lee en el bus no coincide con el que ha transmitido, es que se ha producido un error, así que se marca un "Error de bit". En el siguiente tiempo de bit el nodo transmisor envía una trama de error, entonces, la trama fallida será reenviada después del espacio de intermisión.
- "Bit stuffing" (OSI Layer 1): Como ya se pudo ver, este error detecta si dentro del área codificada por el método del bit stuffing, existen seis bits consecutivos del mismo nivel. El nodo que detecte esto, enviará una trama de error justo en el tiempo de bit siguiente al de detectar el sexto bit del mismo nivel. La trama fallida será retransmitida después del espacio de intermisión.
- CRC (OSI Layer 2): Tanto este método para detectar errores como los dos siguientes (ACK y Frame check) se realizan mediante el chequeo de la misma trama de datos a través de los campos correspondientes. El CRC es un campo de las tramas que contiene un código de redundancia cíclica que comprueba si hubo errores en la recepción del mensaje. Con el CRC podemos detectar errores aleatorios en hasta 5 bits o una secuencia seguida de 15 bits corruptos.

Si el CRC que se calcula en el nodo receptor no coincide con el CRC enviado (el que contiene la trama) se genera un error de CRC.

- ACK error (OSI Layer 2): Otro campo que se encuentra en la trama. En esta ocasión se trata de dos bits que indican si el mensaje fue recibido satisfactoriamente. El nodo transmisor manda el ACK en recesivo esperando a que el nodo receptor lo sobrescriba en dominante, de lo contrario se le considera una trama corrupta y lo retransmitirá. Así, si el transmisor detecta un ACK positivo, es decir, un bit dominante durante el campo ACK, sabrá que al menos un nodo ha recibido correctamente el mensaje.

- Frame check (OSI Layer 2): Algunas partes de los mensajes en CAN tienen formas fijas. Estas zonas son: el delimitador de CRC, el delimitador de ACK, EOF y el IFS. Los bits de estas zonas deben ser recesivos. Si algún controlador CAN detecta que alguno de estos bits es dominante se generará un error de forma "Format error".

Si uno o más errores se detecta por al menos un nodo, usando alguno de los mecanismos arriba expuestos, la transmisión se interrumpe mediante el envío de una trama de error "frame error". Esto evitará que otros nodos acepten el mensaje, lo que garantiza la coherencia de datos en todo la red. Después de la transmisión de la trama de error, el nodo intentará automáticamente el envío del mensaje fallido tan pronto como el arbitraje de la red se lo permita.

Por otro lado, cada nodo contiene dos contadores de error: el Transmit Error Counter (TEC) y el Receive Error Counter (REC). Hay varias reglas que gobiernan el modo en que estos contadores deben incrementarse y/o decrementarse. En esencia, si un nodo transmite una "Trama de Error" el TEC se incrementa en 8 y los nodos que reciben una "Trama de Error" incrementan en 1 el valor del REC. Si un nodo transmite una trama correctamente, el TEC decrementa y si un nodo recibe una trama correctamente, su REC decrementa. Inicialmente los nodos están en estado de "Error Activo". Cuando uno de sus dos contadores llega a superar el valor 127, el nodo pasa a estar en estado de "Error Pasivo", lo que quiere decir que deja de transmitir tramas de error activas. En el caso de que el contador de error TEC llegase a 255, el nodo quedaría anulado y el transceiver se deshabilitaría.

En resumen y como vemos en la Figura 3.6, podemos encontrarnos en 3 estados distintos:

- Error Activo, en este estado el nodo participa en la comunicación normalmente y en caso de detectar un error trasmite la trama correspondiente para señalizarlo.

- Error Pasivo, aquí el nodo aún puede comunicarse pero no generaría tramas de error activas, tan solo podría enviar tramas de error pasivas.

- Anulado, el nodo queda deshabilitado y no participa en la comunicación.

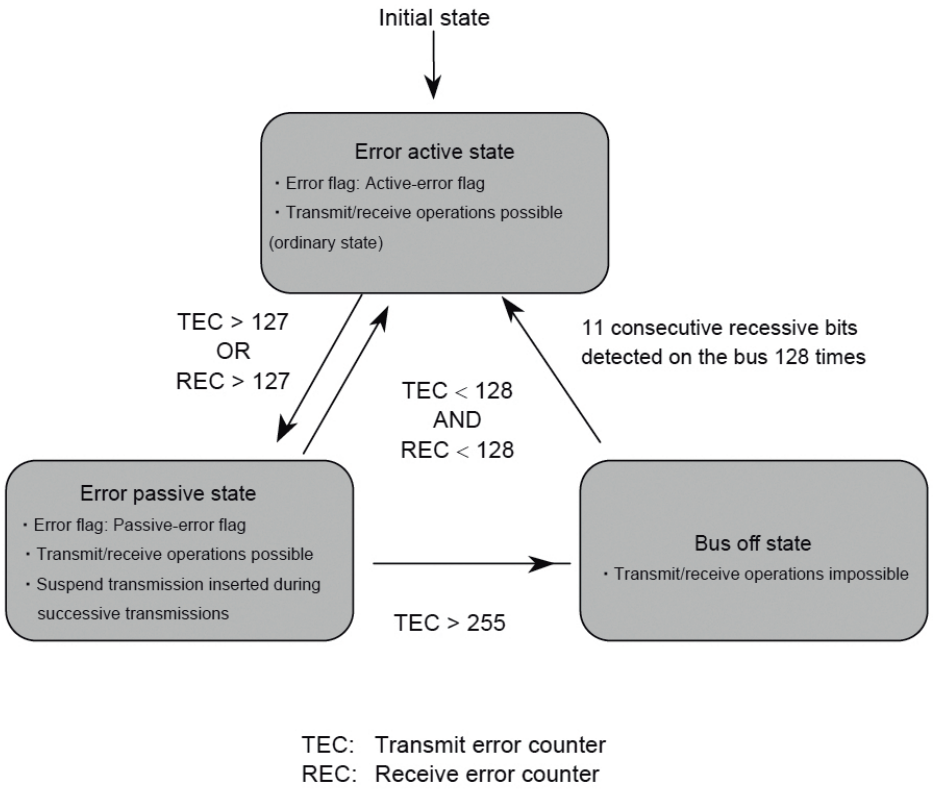


Figura 3.6: Estados de error en un Nodo CAN

3.6. Un nodo CAN

Una red CAN se compone de una serie de dispositivos a los que denominaremos "nodos", conectados a través de un bus serie. Su forma de transmitir como ya sabemos es en broadcast y por tanto, el nodo tiene que tener cierta "inteligencia" para discernir entre mensajes que le son útiles y los que no. Así que un nodo CAN se compone de la siguiente estructura fundamental:

- "Host-Processor": Elemento que se encarga de la "comprensión" de los mensajes recibidos y la elección de los mensajes a enviar.
- "CAN-Controller": Se encarga de la recepción y envío de los mensajes. Básicamente su función al enviar, es almacenar la trama a transmitir e ir mandando los bits de la trama uno a uno. En la recepción va almacenando los datos de la trama bit a bit y una vez completa interrumpe al Host-Processor.
- "Transceiver": Suele estar integrado junto al CAN-Controller, únicamente se encarga de ajustar los niveles lógicos entre el CAN-Controller y el bus físico. También implementa algunos circuitos de protección para aislar en la medida de lo posible al CAN-Controller.

En las especificaciones de CAN no encontraremos un dato que haga referencia a la cantidad de nodos que soporta una red, por tanto debe ser una característica

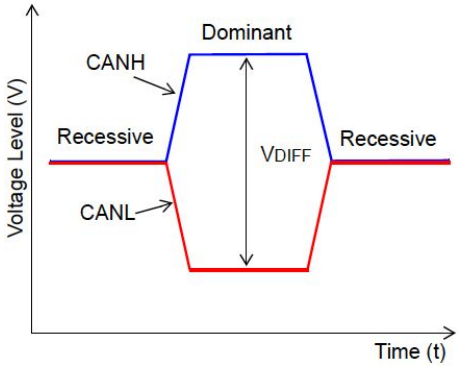


Figura 3.7: Señal diferencial en el Bus

a estudiar a la hora de seleccionar un transceiver, porque dependiendo del modelo este podrá soportar un número distinto.

3.7. La capa física de CAN

La capa física es responsable de la transferencia de bits entre los distintos nodos que componen la red. Define aspectos como niveles de señal, codificación, sincronización y tiempos en que los bits se transfieren al bus.

3.7.1. Características eléctricas del bus

CAN únicamente especificaba dos estados lógicos, recesivo y dominante y no entraba a especificar otras características técnicas importantes como los niveles eléctricos de la señal, sincronización, tiempo de bit o representación. Fue posteriormente, con el estandar ISO11898, donde se definió en profundidad la capa física con la representación de estos estados o bits con una tensión diferencial. Esta ISO fue creada originalmente para la red CAN de alta velocidad en las comunicaciones para vehículos y su objetivo es asegurar la compatibildad entre los transceivers.

La especificación CiA complementó las definiciones respecto al medio físico y conectores. Los nodos conectados al bus interpretan dos niveles lógicos:

- Dominante: la tensión diferencial (CAN-H - CAN-L) es del orden de 2'0 V con CAN-H = 3'5 V y CAN-L = 1'5V(nominales).
- Recesivo: la tensión diferencial (CAN-H - CAN-L) es del orden de 0V con CAN-H = CAN-L = 2'5V (nominales).

Otras de las particularidades dignas de destacar sobre las redes CAN es que si en algún momento alguno de nuestros cables sufre un accidente, ya sea porque se corte o por una derivación a masa, nuestro sistema automáticamente usará la otra línea referenciada a masa para seguir transmitiendo. Por ejemplo, en caso de que el

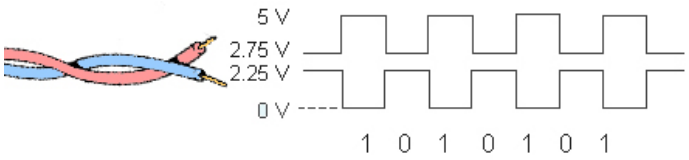


Figura 3.8: Par trenzado

CAN_H se derivase a masa o se cortase, nuestro sistema usaría el CAN_L referenciado a masa para seguir transmitiendo los mensajes a los demás nodos. Teniendo en cuenta que todo el chasis de un automóvil está conectado a masa y que cada uno de los elementos electrónicos están referenciados a dicha masa, resulta una gran ventaja a tener en cuenta.

3.7.2. Cables

Sin duda, una de las principales ventajas que ofrece una red CAN, es la poca cantidad de cobre que necesita para ser implantada, con tan solo un par de cables trenzados y en algunos casos apantallado, podemos conectar todos nuestros nodos. La información por estos cables viaja en modo diferencial, dándole a nuestro sistema una gran robustez frente a interferencias electromagnéticas y eliminando la necesidad de extender un cable de tierra a lo largo del bus, con el consiguiente ahorro.

Según el tipo de aplicación habrá que seleccionar un tipo de cable para la creación del bus y el conexionado de los nodos a este. En la mayoría de los casos se usan cables de 120Ω de impedancia característica y blindados debido a que principalmente se buscan conexiones muy robustas y con amplio margen al voltaje en modo común. Por ejemplo, en entornos muy agresivos donde exista una fuerte presencia de ruido eléctrico que pueda influir en nuestras transmisiones, podremos usar un cable del tipo Belden 3105A. Si por el contrario, trabajamos con distancias cortas, en un entorno con bajo ruido electromagnético, podríamos usar un barato par trenzado telefónico.

3.7.3. Terminaciones de línea

Uno de los problemas que podemos encontrar en las redes CAN son los reflejos y que el propio cable del bus se convierta en una antena, por ello es necesario cerrar las líneas con elementos terminadores, estos son unas simples resistencias que se calculan de forma empírica dependiendo del número de nodos y la longitud del cable. Estas resistencias se ponen al final y al principio de la línea puenteando el CAN-H y el CAN-L.

Usualmente se usan resistencias de 120Ω, coincidiendo de esta forma con la impedancia característica del par trenzado usado para el bus, al coincidir dichas impedancias se maximiza la transferencia de potencia y se disminuyen la ondas reflejas, este concepto es conocido como “Impedance matching”.

3.7.4. Longitud del bus vs Velocidad

Estos dos parámetros son inversamente proporcionales y debemos tenerlos en cuenta, a mayor longitud menor velocidad de transferencia y viceversa. Lo normal es tener buses de longitud corta, para así obtener un funcionamiento óptimo y conseguir la máxima transferencia soportada por el bus CAN, llegando a velocidades de hasta 1 Mbps en distancias menores de 30 metros. Sin embargo, si en una aplicación particular se usan largas longitudes de bus, aunque la velocidad se verá muy afectada, no debería suponer mayor problema, llegando con una tasa 10 Kbps en distancias máximas de 5 kilómetros. A continuación vemos una tabla orientativa de la velocidad del bus en función de la longitud:

Velocidad (Bit/seg)	Tiempo de bit (useg)	Longitud Max. (m)
1Mbps	1	30
800 Kbps	1,25	50
500 Kbps	2	100
250 Kbps	4	250
125 Kbps	8	500
50 Kbps	20	1000
20 Kbps	50	2500
10 Kbps	100	5000

Tabla 3.1: Velocidad y distancia en CAN

Con el objetivo de ser conservadores y 'asegurar' el buen funcionamiento del bus podemos usar una pequeña regla,

$$\textit{Velocidad de transmisión(Mbps)} \times \textit{Longitud del Bus(m)} \leq 50$$

3.7.5. Conectores

No existen conectores definidos en el estandar, aunque deberían tener unas características equivalentes a la línea del bus y a sus terminadores, de forma que no afecten a los parámetros de funcionamiento. Sin embargo, si que existen varias especificaciones sobre conectores y cables en niveles más altos de protocolo, tales como CANopen o DeviceNet, que son ya niveles de aplicación.

3.7.6. Temporización y sincronización de bit

Otra característica importante es que usa codificación Non-Return to Zero (NRZ), lo que quiere decir que el voltaje no vuelve a cero entre bits consecutivos del mismo valor, utilizando el bit stuffing para evitar posibles problemas, que inserta un bit complementario tras 5 bits iguales, además de una cuidadosa temporización de bit.

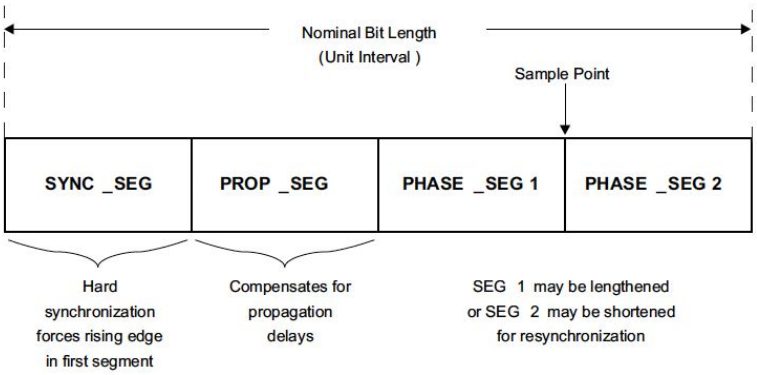


Figura 3.9: Temporización de un bit CAN

Todos los dispositivos en una red CAN tienen su propio reloj, generalmente un oscilador de cuarzo, que pueden tener periodos ligeramente diferentes. Pese a ello, la velocidad de transmisión se ajusta para que sea la misma en todos los dispositivos aunque tengan frecuencias de oscilación diferentes. Al no ser estas frecuencias totalmente estables, los dispositivos se resincronizan con la corriente de bit.

Como vemos en la Figura 3.9, cada bit CAN se divide en cuatro segmentos y el muestreo se realiza aproximadamente al 75 %. El primer segmento es el de sincronización (SYNC_SEG), es el tiempo en el que el bit recesivo pasa a dominante. Todos los nodos del bus sincronizan con los flancos de subida. El segundo segmento, el de propagación (PROP_SEG), está diseñado para compensar el retraso físico de la red. El tercer y cuarto segmento (PHASE_SEG1 y PHASE_SEG2) son usados para la resincronización. El valor del bit es muestreado inmediatamente después del primer segmento de fase (PHASE_SEG1).

Para analizar el contenido de la línea sería necesario usar un osciloscopio digital con dos canales, memoria y un ancho de banda de 20 MHz. En la Figura 3.10 podemos ver una medida en un bus CAN, el osciloscopio usado es de 100MHz. Aunque en este caso los niveles de tensión del CAN-H y el CAN-L son distintos de los comentados, el concepto es el mismo. En color azul podemos ver el CAN-H y en el rojo el CAN-L.

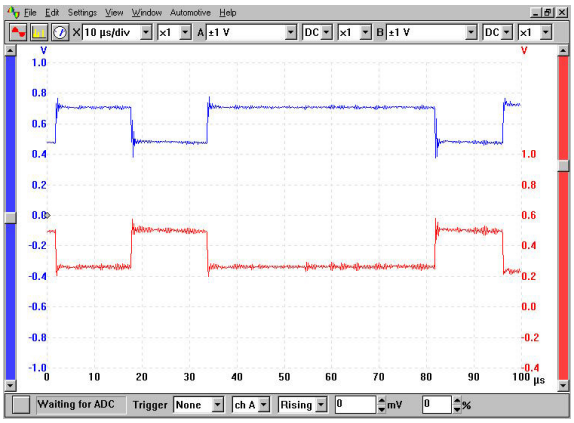


Figura 3.10: Forma de Onda

Capítulo 4

Hardware

4.1. Introducción

Como ya hemos comentado un nodo CAN se constituye con un mínimo de 3 elementos, un Host Processor, un CAN-Controller y un Transceiver. Y por tanto, siguiendo uno de los objetivos del proyecto, vamos a diseñar un nodo CAN para Arduino. En este caso, el propio Arduino será el que desempeñe el papel de Host-Processor y nuestro shield tendrá que aportar los otros dos elementos restantes, el CAN-Controller y el Transceiver.

El mercado nos ofrece algunas soluciones integradas que vamos a analizar con el fin de escoger la que mejor se adapte a nuestras necesidades.

Como el proyecto trata de Arduino y CAN bus, he decidido que el nombre comercial para el Shield que diseñaremos será "**ArCan**' que lógicamente, viene de la asociación ARduino+CAN.

4.2. Alternativas para un nodo CAN

Inicialmente, a la hora de plantear el diseño de nuestro nodo CAN, damos un vistazo a las posibles alternativas que encontramos en el mercado y debemos tomar la primera decisión, ¿Qué dispositivos serán más aconsejables en nuestro proyecto?. En primer lugar no encontraremos una solución del tipo System On Chip (SOC), es decir, que tanto el controlador CAN como el transceiver estén integrados en el mismo silicio. Por otro lado, encontramos que algunas versiones de microcontroladores traen integrados como periféricos el CAN-Controller. Finalmente, también podemos encontrar todos los dispositivos por separado, siendo en primera instancia la opción más indicada, puesto que la Host-Controller será la que integre Arduino y si elegimos un microcontrolador con el periférico CAN integrado incrementamos el coste

de forma absurda.

Por tanto y según lo mencionado, vamos a buscar en el mercado las opciones que ponen a nuestra disposición, empezando por el CAN-Controller.

- SJA1000
- SAE 81C90/91
- Intel 82527
- Bosh CC770
- MCP2515

Los 4 primeros se controlan a través de un bus paralelo de 8 bits multiplexando datos y direcciones. Además, no hay que olvidar las señales de control, RST, CS, RD, WR, INT, ALE, siendo necesario usar 14 pines I/O y por tanto dejando a Arduino prácticamente sin recursos para poder conectar otros elementos. Aunque la programación no parece compleja, es inaceptable agotar tantos recursos 'externos' de Arduino, recordemos que tan solo posee 19 pines de entrada y salida.

Según las características y los precios, Microchip nos ofrece una solución completa en dos integrados. El CAN-Controller MCP2515, a diferencia de sus alternativas en el mercado, usa un bus SPI para establecer la comunicación con el Host-Controller y Arduino cuenta con un periférico del mismo tipo que nos facilitará en gran medida la programación y con lo que tendremos que destinar tan solo 4 pines para el control del bus CAN. Aunque la elección del Transceiver no es tan importante, puesto que la mayoría tienen características muy similares, por precio y por la propia recomendación del datasheet del MCP2515, hemos seleccionado el MCP2551, este transceiver nos permite una conexión de hasta 112 nodos y cumple con la norma ISO11898, así que está preparado para redes CAN de alta velocidad.

Para realizar los prototipos usaremos el servicio de SAMPLES de Microchip, que nos permitirá montar los prototipos prácticamente sin coste.

4.3. El CAN-Controller: MCP2515

Implementa la versión CAN V2.0B a 1Mb/s, así que es compatible tanto con "Standard Message Format" de 11 bits como con "Extended Message Format". Entre las características más interesantes que incorpora es su interfaz Serial Peripheral Interface (SPI) a 10MHz, que facilitará mucho el desarrollo del software, pudiendo escribir y leer cualquier registro del dispositivo usando los comandos estandars SPI.

Además, entre otras cosas, cuenta con dos máscaras y seis filtros de 29-bits con el objetivo de rechazar mensajes no deseados, de esta forma ahorramos el procesado de filtrado en Arduino, dejando así, la CPU más liberada para dedicarla a otras funciones. Tiene 2 buffers de recepción, para evitar una pérdida de mensajes si el Host-Controller no ha sido capaz de procesarlo con suficiente velocidad y 3 buffers

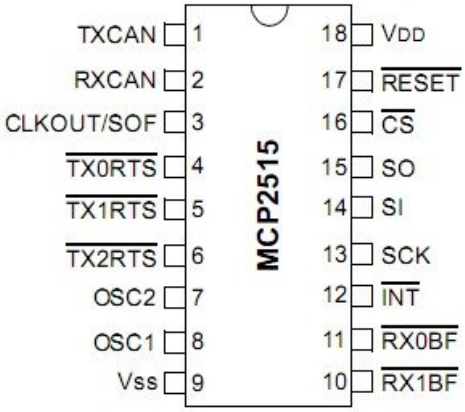


Figura 4.1: MCP2515

de transmisión, donde almacenará los mensajes a enviar hasta que sea posible su transmisión, liberando de esta tarea al Host-Controller.

Además, con el objetivo de poder realizar una gestión más eficiente, el dispositivo cuenta con una patilla de interrupción de propósito general, pudiéndose configurar para que se active cuando se recibe un mensaje, de esta forma evitamos estar haciendo un polling constante a través del SPI para conocer si existe algún mensaje en los buffers de recepción, reduciendo así el overhead en el Host-Controller. Como vemos en la imagen, también tiene 3 patillas para activar la transmisión, TXnRTS, siendo el uso de estas opcionales, pues también se puede forzar la transmisión a través del SPI.

Por último, mencionar que Low-Power Complementary Metal Oxide Semiconductor (CMOS) es la tecnología de fabricación, operando en un rango de tensión de 2,7V y 5,5V, que lo hace perfectamente compatible con Arduino, y con un consumo en activo de 5mA y tan solo 1uA en el modo dormido.

4.4. Controlador y transceiver CAN

Sin duda lo más importante de nuestro circuito, como ya sabemos los elegidos han sido las soluciones ofrecidas por Microchip. Como controlador CAN montaremos un MCP2515 y como transceiver el MCP2551.

Añadiremos a nuestro circuito, debido al bajo coste y trabajo de implantación, la resistencia para el cierre del bus, seleccionando su uso, en caso necesario, con un simple Jumper, esto nos ahorrará buscar soluciones o mecanismos externos quedando todo perfectamente integrado en nuestro nodo y facilitando la tarea de instalación.

El conector para el bus será un simple bloque atornillable con 3 conexiones, GND, CAN-H y CAN-L, sin duda es la forma más sencilla de evitar complicaciones y posibles limitaciones para los usuarios finales, que no deberán usar ningún conector especial para el uso del bus, simplemente atornillar los cables.

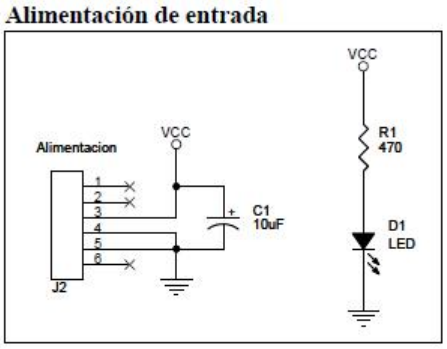


Figura 4.2: Fuente de alimentación ArCan

4.5. Fuente de alimentación

Como todo circuito electrónico es necesario dimensionar los diferentes componentes con el fin de proporcionar un funcionamiento estable y robusto a fin de evitar fallos electrónicos. El diseño que nos compete no es necesariamente complicado gracias a los circuitos integrados que usaremos. Para empezar necesitamos conocer los niveles de tensión y corriente con el que trabajarán el CAN-Controller y el Transceiver, pues en base a ello debemos diseñar la fuente de alimentación que nos asegure un funcionamiento óptimo y robusto contra posibles fallos de tensión.

Por tanto, vamos a empezar con el diseño de la fuente de alimentación. Según las características del MCP2515 es un circuito de tecnología Low-Power CMOS y opera en unos rangos de tensión de 2,7V y 5,5V con un consumo máximo de 10mA. A su vez, el transceiver MCP2551 es bastante más exigente en lo que a rango de tensiones y corrientes se refiere, funcionando correctamente entre los 4,5V y 5,5V y con un máximo de 75mA en caso de bits dominantes.

Analizando la fuente de alimentación de la que dispone Arduino y teniendo en cuenta que el circuito que deseamos diseñar no tiene cabida sin él, vemos como opción más recomendable y a la vez más económica el uso de esta, puesto que goza de un regulador NCP1117ST50T3G capaz de otorgar una tensión de 5V y soportar una corriente máxima de 1A, más que suficiente para alimentar tanto al shield como al propio Arduino. Al parecer en Arduino se prevee una doble implantación para el regulador de tensión, usando en ocasiones el MC33269 que también proporciona 5V aunque la corriente que es capaz de suministrar se ve ligeramente mermada con respecto al anterior, llegando a los 800mA.

Vemos en un primer análisis, que la alimentación integrada en Arduino es sobradamente capaz de soportar nuestro circuito, desgraciadamente en la documentación disponible de Arduino no se menciona unos requisitos mínimos en el diseño de la fuente de alimentación, dejando cierta incertidumbre en versiones futuras, aunque siguiendo la larga trayectoria del proyecto, podríamos suponer que al menos siempre será un regulador de 5V capaz de entregar un mínimo de 500mA.

Además, para darle estabilidad y eliminar posibles picos de tensión que puedan alterar el funcionamiento normal de nuestro circuito, añadiremos algo de carga capacitiva.

4.6. Oscilador

Según el datasheet, el oscilador para el MCP2515 puede diseñarse tanto con un cristal de cuarzo como con un resonador cerámico. Sin duda, los resonadores cerámicos son componentes más económicos que los de cuarzo, aunque al ser construidos con semiconductores sus tolerancias son mayores, siendo por lo general el 1 % frente a los 30-50ppm de los cristales.

También hay que tener en cuenta que los resonadores, pese a ser ligeramente más económicos y llevar integrados los condensadores, son menos comunes, por lo que en principio haremos el desarrollo usando un cristal de cuarzo, por su facilidad para ser adquiridos y su precisión. Hay que tener en cuenta que para que el MCP2515 pueda funcionar con su máximo baudrate, 1Mb/s, sería necesario montar un oscilador de, al menos, 16MHz, con un límite de 20MHz. Los condensadores vendrán determinados por el fabricante del oscilador de cuarzo, siendo una guía la tabla que podemos encontrar en el datasheet del MCP2515.

4.7. Reset

El MCP2515 tiene dos formas para realizar el reset, por un lado existe el Hard-Reset, que consiste en poner a LOW el pin de RESET del circuito integrado y por otro se puede realizar a través de un comando 'reset' desde el SPI. En cualquier caso, creo que es recomendable añadir un botón de reset a fin de poder reiniciar el sistema en cualquier momento de forma manual y sencilla, además, puesto que el diseño que nos atañe a realizar es un shield, existiendo la posibilidad que el propio botón de reset que tiene Arduino quede tapado por este. Sin duda, la forma correcta de realizar un reset del circuito sería también resetear la propia placa de Arduino al mismo tiempo, para ello podemos usar la señal de reset que encontramos en el conector In-Circuit Serial Programming (ICSP), de este modo, con una simple pulsación del nuestro botón de reset reiniciaremos todo nuestro sistema.

4.8. Buffer de recepción

Nuestro CAN-Controller posee dos pines con el objeto de señalizar cuando los buffers de recepción se encuentran llenos. Cada pin corresponde a un buffer y cada buffer es capaz de almacenar un mensaje CAN completo. A fin de poder ver el tráfico en la red CAN y como un rápido mecanismo para la búsqueda de posibles errores, es conveniente usar estas señales para activar unos leds, que de forma rápida y muy visual nos darán la información sobre el estado de los buffers de recepción. Un detalle a tener en cuenta, es que estos pines son activos a nivel bajo, lo que quiere decir que cuando el buffer esté lleno, impondrá un 0 lógico que correspondería a GND. Dicho esto, y a fin de que los leds solo se encuentren iluminados cuando el buffer esté lleno, debemos hacer la conexión entre Vdd y el pin correspondiente, RX0BF y RX1BF, añadiendo además una resistencia que limite el consumo del led.

4.9. Sensores externos

Es conveniente y dado el carácter tan general de Arduino, proporcionar unos conectores atornillables para la posible conexión de sensores y/o actuadores. También tendremos en cuenta proporcionar alimentación para los posibles sensores, trazando para ellos las líneas de VCC y GND de la propia placa Arduino hacia unos conectores atornillables.

4.10. Protección por sobrecargas

Aunque la información sobre el consumo de Arduino no se encuentra disponible oficialmente y detallada de forma teórica, sí que podemos hacer algunas pruebas empíricas con las que se concluyen que el consumo ronda los 50mA. Por tanto y teniendo en cuenta el del circuito CAN que pretendemos diseñar, unos 85mA, tendríamos un consumo 'interno' de 135mA, quedando hasta el límite de la fuente de alimentación que analizamos anteriormente, unos 665mA disponibles. Por tanto, para evitar posibles sobrecargas por parte del usuario debido a algún problema en los sensores/actuadores conectados a esta línea de alimentación, usaremos un fusible reseteable o también conocido como Poliswitch de 500mA.

Este tipo de fusibles tienen la ventaja que se rearman automáticamente cuando se enfrían, siendo por tanto un estupendo mecanismo de protección sin ningún tipo de mantenimiento, al contrario de sus hermanos convencionales, que había que cambiarlos cuando se fundían.

Hay que remarcar que el propio Arduino posee uno de estos fusibles, aunque únicamente lo usa cuando la alimentación de corriente se produce a través de la entrada Universal Serial Bus (USB), dejándolo totalmente desprotegido en caso de una alimentación externa a través del conector Jack, es por ello, por lo que me parece conveniente añadir un mecanismo de seguridad que además de proteger a los sensores también proteja a la propia fuente de alimentación de Arduino, aunque el regulador de tensión incorpore internamente mecanismos de protección por limitación de corriente y por exceso de temperatura.

4.11. Mejoras y protección para el Bus

El bus CAN suele necesitar protección que ayude a que los nodos sobrevivan en los ambientes más violentos en lo que a ruido, transitorios y tensión en modo común se refiere. No hay que olvidar que CAN está pensado para la automoción y la industria donde podemos encontrar grandes cargas capacitivas e inductivas, fuentes de tensión conmutadas, el balastro de un flourescente, ruido producido por relés o por el arranque de motores, etc. Todo este tipo de fuentes externas de interferencias electromagnéticas pueden causar, sin la debida protección, que nuestros nodos pasen a mejor vida.

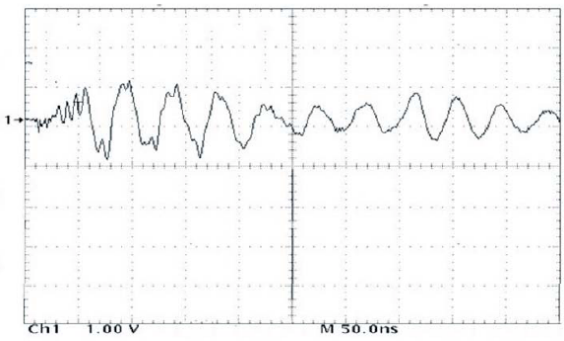


Figura 4.3: Ruido acoplado en el bus con par trenzado

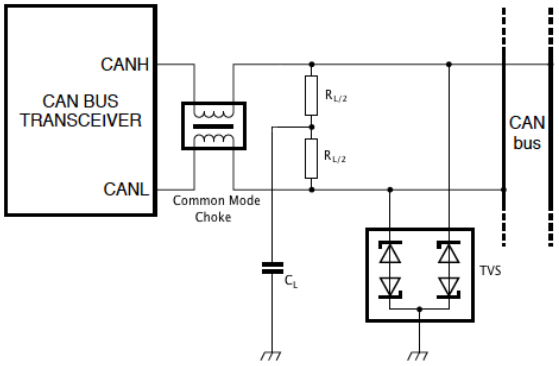


Figura 4.4: Circuitos de protección para el bus CAN

Aunque los transceiver incorporan algunos mecanismos de protección, dependiendo de la aplicación podrían ser insuficientes. Recordemos que la norma ISO11898 especifica una protección del tipo "Short Cirtcuit" de $-3V$ a $32V$ para sistemas de $24V$ y de $-3V$ a $16V$ para los sistemas de $12V$, esta protección entra en funcionamiento principalmente cuando se conecta el nodo al bus, impidiendo, que por culpa de una diferencia de potencial inicial, se pueda dañar el circuito. También se especifica una protección mínima para los transitorios que oscila entre $-150V$ y $100V$ y una tensión en modo común desde $-2V$ a $7V$. Por lo general, estas cifras son ligeramente mejoradas por la mayoría de transceiver del mercado, no obstante, es posible que aún así, en una aplicación industrial o de automoción sea necesario mejorar dichos valores a fin de dar la robustez necesaria al conjunto.

Como ya sabemos, la señal diferencial CAN es el resultado de $CAN_H - CAN_L$, para un bit recesivo el valor típico es de $0V$ mientras que para uno dominante es de $2V$, como se especifica en la ISO11898-2. Los problemas ocurren cuando el ruido que se acopla a la línea del bus está cercano a estos valores. No es descabellado pensar que este nivel de ruido puede añadirse debido al efecto "antena" que podría producir el cable que usamos para construir el bus. En la Figura 4.3 podemos observar el ruido acoplado en un bus que usa par trenzado.

En la Figura 4.4 vemos un ejemplo con todos los mecanismos de protección más usados en el mundo CAN, con los que se mejora notablemente su robustez. Pasaremos a analizar como funcionan y porqué es interesante considerar estas medidas de protección extras.

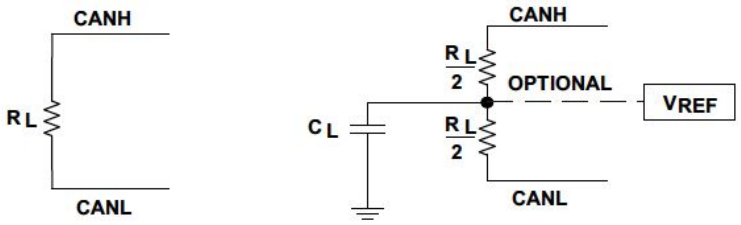


Figura 4.5: Terminaciones: Standard y Split

4.11.1. Terminación en Split vs Estandard

Ya hemos mencionado el mecanismo de cierre o terminación del bus CAN habitual, que consiste en montar unas resistencias, usualmente de 120Ω en los extremos del bus, pero existe otro tipo de terminación discretamente más compleja que presenta mejoras en cuanto a interferencias y emisiones electromagnéticas conocido como "Split Termination". Tal y como podemos observar en la Figura 4.5, consiste en usar dos resistencias del mismo valor y un condensador, con ello se consigue hacer un filtro paso bajo de forma que las componentes de alta frecuencia se deriven a tierra. Calcular el valor correcto del condensador es tan simple como aplicar la fórmula para un filtro paso bajo de primer orden, teniendo en cuenta que las dos resistencias forman un circuito paralelo con respecto al condensador. En el estandar ISO11898-5, para redes de alta velocidad, se especifica un $C_L = 4,7nF$ para unas $\frac{R_L}{2} = 60\Omega$, de esta forma se crea un filtro con $3dB$ de caída en $1,1Mbps$. Aplicando los cálculos teóricos:

$$R_p = 60\Omega||60\Omega = 30\Omega$$

$$C_L = 4,7nF$$

$$F_c = \frac{1}{2\pi C_L R_p} = 1128758,4Hz \simeq 1,1Mbps$$

Para ver el comportamiento en frecuencia del filtro que nos atañe y verificar que los cálculos son correctos, usaremos el Matlab con las funciones del "System Control Toolbox" que nos facilita el cálculo y la representación de filtros. Para ello, creamos un pequeño script como sigue,

```
R = 30;
C = 4.7e-9;
Y = [1];
X = [R*C 1]
filtroRC = tf(Y,X)
option = bodeoptions;
option.FreqUnits = 'MHz';
option.PhaseVisible = 'Off';
bode(filtroRC,option)
```

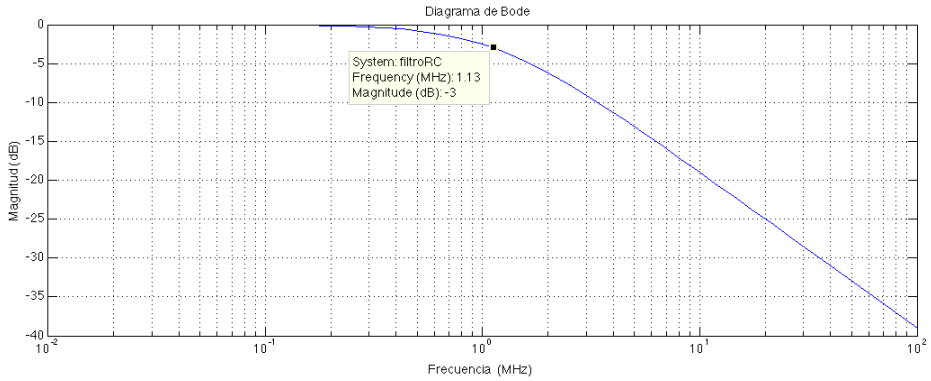


Figura 4.6: Diagrama de Bode para el fitro de la terminación Split

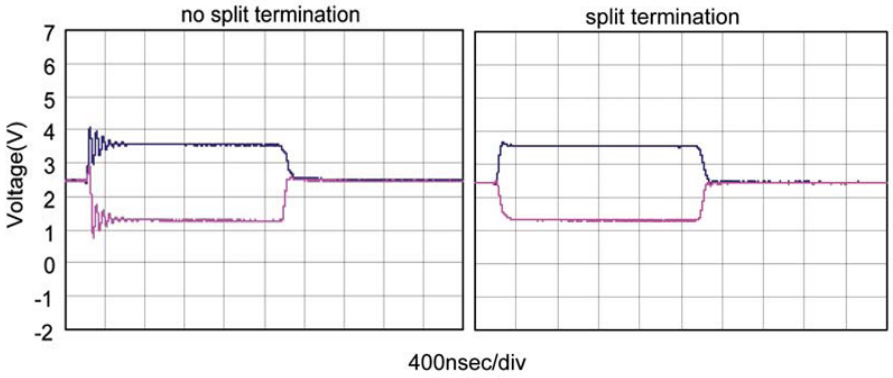


Figura 4.7: Diferencia entre terminación estandard y Split

Generando como resultado el diagrama de Bode de nuestro filtro representado en la Figura 4.6, con el que podemos comprobar rápidamente que el comportamiento en frecuencia que tendrá, es el esperado.

En la práctica, como vemos en la Figura 4.7, podemos observar a través de las capturas del oscilocopio, la señal tan limpia que presenta un bus con terminación Split con respecto a uno con la terminación estandard.

4.11.2. Protección contra el modo común

La tensión en modo común o "Ground Offsets" se crea cuando existe una diferencia de potencial entre la referencia del circuito emisor y el receptor. Aunque nuestro bus CAN es diferencial, hay que entender que una tensión en modo común con valores altos podría destruir nuestro circuito.

Existen dispositivos electrónicos pasivos que solucionan el problema generado por el ruido en modo común que llega a los nodos, generalmente, para este tipo de problemas se usan los Common-Mode Chokes. Este tipo de dispositivos son capaces de impedir el paso de la corriente en modo común, mientras que no pone ninguna oposición a la diferencial.

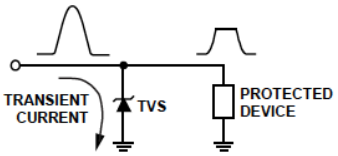


Figura 4.8: Funcionamiento básico de un TVS

4.11.3. Protección contra transitorios

Este tipo de fenómeno se suele producir al conectar las líneas del bus, de forma que producen un pico de tensión que de no ser suprimido, podría destruir el circuito. Como dispositivos de protección económicos y con alto nivel de integración, podemos encontrar en el mercado los diodos Transient Voltage Suppression (TVS), que se encargan de absorber los picos de tensión que se puedan producir. En la Figura 4.8 podemos ver el funcionamiento de un TVS de manera sencilla para entender mejor el mecanismo de protección. Hay que tener en cuenta, que al situarse en las líneas de transmisión deberán estar cuidadosamente seleccionados, pues podrían influir y distorsionar la señal. Existen dos parámetros básicos a tener en cuenta en la selección de un TVS para CAN, el V_{RWM} que es la tensión máxima en el que el diodo se comporta prácticamente como un circuito abierto, circulando a través de él una pequeñísima corriente del orden de nA, y la capacidad asociada al diodo C_d que debería ser lo más pequeña posible para evitar que distorsione nuestra señal. Para redes de alta velocidad, este valor no debería superar los $30pF$. Por ejemplo, podríamos usar el $PESD2CAN$ de NXP o el $NUP2105L$ de $On\ Semiconductor$.

4.12. Prototipos: Diseño, construcción y pruebas

Una vez estudiado el hardware y los componentes a usar, hay que realizar un prototipo con el que hacer unas pruebas simplificadas para verificar el funcionamiento práctico. Los pasos a seguir para tener éxito en esta fase han sido 3, e intentaré detallarlos a continuación.

4.12.0.1. Esquemático

El diseño del esquemático ha sido realizado en OrCad Shematic 9.2 por mi amplia experiencia en él. Ha sido necesario añadir los componentes propios del CAN bus y buscar la forma correcta de conectarlos con Arduino. Se ha diseñado intentando separar los bloques según su funcionalidad y por tanto facilitando la comprensión de cada elemento para el usuario, además, esta forma de hacerlo ayuda a la hora de buscar errores y fallos. Aparentemente se trata de un diseño sencillo aunque ha sido necesario estudiar en profundidad los recursos disponibles por Arduino y dimensionar los elementos de forma que podamos obtener las máximas prestaciones al menor coste. Por ello, hay elementos opcionales en el montaje que mejoran el rendimiento en entornos ruidosos a costa de un pequeño incremento en el precio.

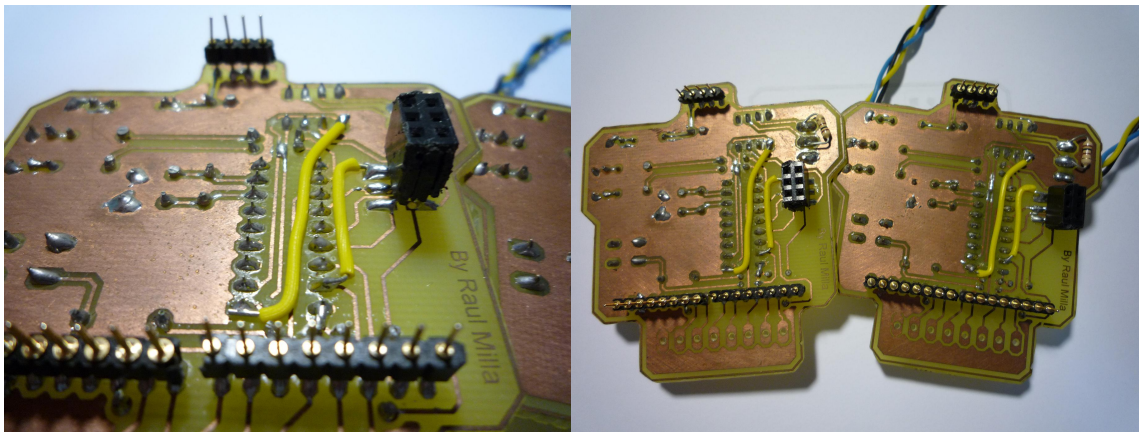


Figura 4.9: Fotos del 1º prototipo, taladro pasante

Componentes específicos como el CAN-Controller o el transceiver, no se encontraban en las librerías generales de OrCad y han sido creados según el datasheet.

4.12.0.2. Diseño PCB

El desarrollo del circuito se ha realizado con OrCad Layout 9.2, por el mismo motivo que el caso anterior. Una vez que tenemos el esquemático hay que empezar con el desarrollo de la placa de circuito impreso o Printed Circuit Board (PCB). Al ser un shield de Arduino tenemos una serie de requisitos en dimensiones y ubicación de ciertos componentes, como por ejemplo los pines que conectarán ambas placas, por tanto es absolutamente necesario que sean los primeros componentes que posicionemos en nuestro diseño. Además, como en el caso del esquemático, no encontraremos todas las huellas o footprints en la base de datos por defecto, así que fue necesaria también su creación a partir del datasheet de los componentes.

4.12.0.3. Fabricación de prototipos

Una vez con la PCB diseñada, el último paso para crear el Shield es su fabricación, inicialmente y para comprobar posibles errores de diseño se han realizado dos prototipos usando una insoladora de construcción casera y placas de circuito fotosensibles. Como vemos en la fotos con referencia a la Figura 4.9 hubo algunos fallos de rutado en la PCB que se corrigieron haciendo cortes en las pistas y realizando las conexiones con cable.

Aunque el circuito con insoladora y placa fotosensible no nos dio muy buen resultado, repetimos el proceso añadiendo la corrección a los primeros errores detectados en la versión anterior. También tuvimos que realizar cambios en la huella del CAN-Controller, porque no encontrábamos la versión Plastic Dual-in-Line Package (PDIP), taladro pasante, en stock, así que nos animamos a usar la versión Surface Mount Technology (SMT). El resultado podemos verlo en la foto con referencia a la Figura 4.10. Aunque la calidad de los prototipos deja que desear, han cumplido perfectamente su misión y nos han permitido comprobar que hemos conseguido

una primera versión que funciona y con la que conseguimos establecer comunicación desde Arduino.

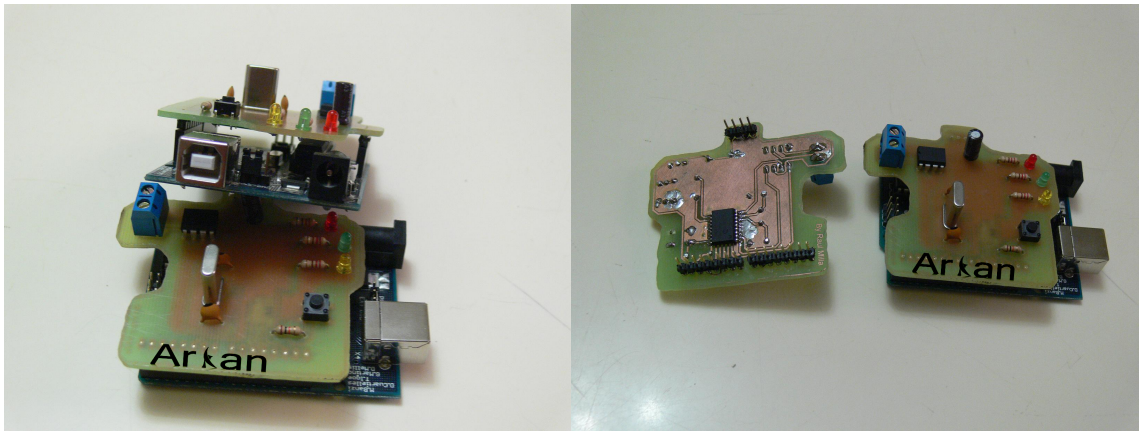


Figura 4.10: Fotos del 2º prototipo, algunos componentes SMD

En las primeras fases, las de fabricación casera, se han empleado únicamente componentes de taladro pasante, salvo en el segundo prototipo porque el MCP2515 no se encontraba en stock con este encapsulado, para facilitar el montaje manual y de tamaño considerable. En los prototipos que se externalizará su fabricación, se han diseñado usando componentes de montaje superficial, Surface Mount Device (SMD), con lo que se consigue un acabado más profesional y permite el montaje automático, aunque las primeras se ensamblarán a mano para bajar los costes de desarrollo y detectar posibles errores.

El resultado final, como vemos en las Fotos referenciadas con la Figura 4.11, es espectacular. Esta versión ha funcionado desde el primer instante y no ha presentado ningún problema en el montaje manual aunque tuviese los componentes SMD. La comunicación con Arduino ha sido instantánea ya que teníamos un pequeño programa de pruebas que escribía los registros necesarios en el MCP2515 para encender los Leds rojo y naranja que vemos en la imagen.

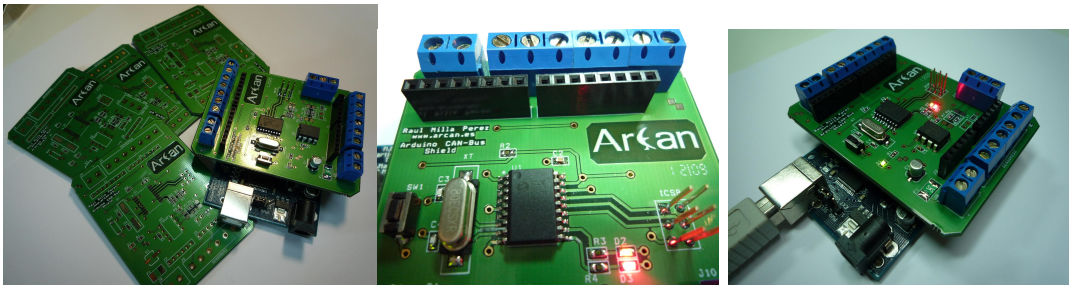


Figura 4.11: Fotos del 3º prototipo, fabricación profesional

Desde el inicio del proyecto se han realizado muchos cambios en el esquemático y en la PCB, añadiendo mejoras, corrigiendo fallos o modificando las dimensiones y encapsulados de algunos componentes a fin de conseguir una placa económica y con componentes de fácil adquisición. Desde la versión del prototipo de fabricación externa se ha llevado un control sobre los cambios que se debían realizar para conseguir un producto de calidad y fácil de trazar en la búsqueda de errores futuros.

En ArCan se ha realizado un control de versiones para poder trazar fallos y corregirlos fácilmente. El código de control que se ha empleado consiste en 2 elementos nombrados de la siguiente forma, RevX_eX. La letra que acompaña a 'Rev', hace referencia a la Revisión del Hardware, es decir, cuando hay elementos en el Hardware que han cambiado o se han añadido/eliminado de la PCB. El dígito que acompaña a la 'e' hace referencia a la edición, suelen ser pequeños cambios que no deben modificar el comportamiento funcional del Hardware. Veamos el desarrollo de algunos cambios en las versiones de ArCan.

RevD_e0 a RevD_e1:
Se modifica el Footprint del SW1, aumentando el diámetro de corona a 0.3mm, esto reduce el coste de fabricación de la PCB.

RevD_e1 a RevD_e2:
Se modifica el esquemático para conectar los led que señalizan la transmisión a VCC ya que la salida del MCP2515 es negada. El problema que existía es que su estado era normalmente iluminado por lo que el consumo de la PCB se disparaba.
Se modifica el rutado de la PCB para optimizar y hacerlo más simple.
Se modifica la huella del MCP2551 y se pone su versión SMD para el montaje automático.

RevD_e2 a RevD_e3
Se modifica el encapsulado de la resistencia de terminación del bus CAN a 1208, la de 0603 no disipa suficiente potencia según los cálculos.
Se modifica la serigrafía del mcp2551, u2
Se cambiar la serigrafía de los leds, led_pwd, led_tx, led_rx

4.12.0.4. Primeras pruebas

En esta fase del desarrollo, el conocimiento que se tiene sobre CAN y el interfaz SPI es relativamente amplio y estamos en capacidad para probar los primeros prototipos.

Algo que retrasó bastante la planificación del proyecto fue un fallo que se encontraba en la placa de Arduino, la versión que estábamos usando no estaba del todo bien diseñada en lo que a la interfaz SPI se refiere, ya que el pin SCK tenía una resistencia en serie además de un led, esto degeneraba demasiado la señal de reloj y no se producían los desplazamientos de bit correctamente en el interfaz SPI del MCP2515. Aunque este fallo en apariencia pueda resultar trivial, cuando uno construye unos prototipos tan artesanos como fueron estos, siempre atribuye el problema al prototipo, ya que el diseño "básico" era bastante sencillo. En la Figura 4.12 podemos ver el problema.

Una vez descubierto el fallo y modificado los Arduinos en posesión para que no presentaran el problema, la primera prueba que se realizó fue configurar los leds de los buffers para que se iluminarán en el momento de la secuencia de inicialización.

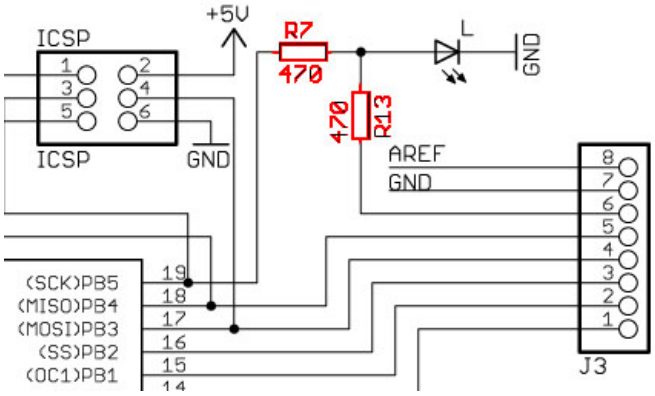


Figura 4.12: Fallo en el pin SCK

Con esta prueba lo que se pretendía era conocer si podíamos modificar los registros del MCP2515. Una vez conseguido, el siguiente paso era configurar y verificar que el valor escrito en los registros del Can-Controller eran los deseados.

Para realizar la prueba y encender los leds, únicamente es necesario escribir el registro BFPCTRL del MCP2515. En este registro es conocido como "Pin Control And Status" y está en la dirección 0x0C de nuestro Can-Controller. En el está mapeado el uso y funcionamiento de los pines RX1BF y RX0BF. A través de los bit correspondientes, ajustaremos la función de estos pines como salida digital con valor de salida '0'. Un pequeño código de ejemplo sería:

```
//En primer lugar seleccionamos ArCan

digitalWrite(ArCan_CS,LOW);

//Mandamos el comando "escribir"

spi_putc(SPI_WRITE);

//Enviamos la "dirección"

spi_putc(BFPCTRL);

//Y el valor que queremos escribir en el registro BFPCTRL

spi_putc(0x0C);

//Por último deseleccionamos ArCan

digitalWrite(ArCan_CS,HIGH);
```

Como vemos, en primer lugar debemos activar ArCan a través del pin definido con la etiqueta ArCan_CS. En segundo lugar escribimos el comando WRITE en el puerto SPI seguido de la dirección de memoria del registro, que en este caso se encuentra definida en la etiqueta BFPCTRL. En tercer lugar escribimos el valor correcto en el puerto SPI para configurar los pines mencionados con la función ya descrita. Para finalizar desactivamos la placa ArCan. El resultado es que tendremos de forma permanente los pines conectados a los leds activados y por tanto estos se encontrarán iluminados.

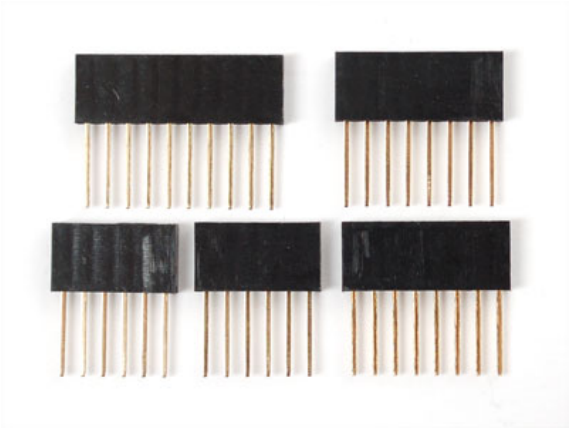


Figura 4.13: Pines específicos para shield

Una vez comprobado que la comunicación se produce correctamente, está todo preparado para enviar señales por el bus CAN, aunque este proceso era bastante más complicado. De forma rápida, puesto que aún no esta desarrollando la librería, hice un pequeño escenario de pruebas con un analizador lógico y un conjunto ArCan. La forma más rápida de comprobar su funcionamiento es que en la secuencia de inicialización, se configura un mensaje CAN manualmente y se procede con al envío de este. Gracias al analizador lógico, que en nuestro caso tenía soporte para CAN bus, fue posible comprobar, que efectivamente, se estaba realizando la transferencia en el bus.

Llegados a este momento, y habiendo comprobado que los prototipos caseros funcionaban correctamente, modifiqué lo necesario y encargué la primera versión a una empresa externa, la RevD_e1. A la espera de los nuevos prototipos de acabado profesional, empecé con el desarrollo de la librería que nos facilitaría la programación y pruebas futuras.

4.13. Costes y búsqueda de componentes

Durante el diseño del circuito se han tenido en cuenta las especificaciones de Arduino para poder dimensionar el Shield a fin de que se adapte a él. El mayor problema que hemos encontrado, es la búsqueda de la tira de pines como los que vemos en la Figura 4.13. El problema es que este tipo de pines, con la longitud tan larga que presentan los espadines, no son estándares y por tanto no se encuentran tan fácilmente. El fin de que el espadín sea tan largo es para permitir el apilado de varios Shields. En los inicios del proyecto, este tipo de pines solo se podían adquirir en http://www.nkcelectronics.com, aunque actualmente, se han vuelto más común, debido al fuerte crecimiento que han experimentado los nuevos desarrollos de Shields para Arduino.

Las placas de circuito impreso han sido compradas en http://www.pcbcart.com. Se realizó un pedido de 50 unidades con un coste total 153$. Dichos costes están divididos tal y como se describe en la Tabla 4.1.

Coste unitario	1,53$
Gastos de Tooling	35,30$
Gastos de envío	35,79$

Tabla 4.1: Tabla de costes de fabricación de la PCB

Es interesante saber que los costes de Tooling hacen referencia a la configuración de las máquinas necesarias para la fabricación del circuito. Este tipo de coste es único y solo se paga la primera vez, siempre y cuando no existan cambios en la placa de circuito. Como vemos, el precio conseguido para la fabricación de tan pocas cantidades es muy bueno. Dividiendo esos costes entre las 50 unidades fabricadas el precio total por unidad sería de 2,95$ lo que supone al cambio actual, 1$ son 0,79€, tan solo 2,20€. Obviamente, para futuras tiradas el precio bajaría aún más, pues como hemos comentado, el coste de Tooling no estaría presente.

Aunque los primeros circuitos han sido montados a mano, en producción real habría que añadir también los gastos de montaje, al igual que con la fabricación, estos gastos incluyen la primera vez el Tooling. En la web http://www.myropcb.com, podemos encontrar un servicio de montaje para pequeñas cantidades con condiciones sobradamente interesantes. La forma de realizar el presupuesto es dependiente del número de pines, con un precio de 0,0125$ por pin SMT y uno de 0,025$ por el de taladro pasante. En nuestro Shield, encontraremos un total de 62 pines SMD y 58 de taladro pasante, lo que haría un total de 2,3$ por el montaje unitario que al cambio actual son 1,76€.

Una vez sabido tanto el coste de fabricación como el de montaje, queda hacer un análisis real de los costes que tendrá los Shields suponiendo una fabricación a pequeña escala con tan solo 50 unidades.

El precio en el que se mueve este tipo de dispositivos oscila entre los 25€ y los 45€, dependiendo de la complejidad y las funciones que implemente. En el momento del desarrollo, empecé su comercialización a 30€ la unidad, aunque después y en vista de la aceptación lo subí a 35€, un precio coherente con las demás alternativas del mercado. A día 30 de Abril de 2013 se han vendido más de 50 unidades.

Hay que tener en cuenta que además del coste del Shield, habrá que sumarle el de un Arduino, que suele rondar los 24€ para así conseguir un nodo CAN totalmente autónomo.

Junto a los costes, habría que señalar el coste del desarrollo, que en este caso abarcaría tanto el diseño del Hardware como el de la librería, este coste es realmente difícil de cuantificar porque durante el proyecto es mucho lo que se ha invertido en aprendizaje y hoy por hoy, el tiempo de desarrollo completo seria mas reducido.

Si quisiéramos concretar el coste humano derivado del proyecto, desde mi punto de vista, un trabajo de este tipo se podría acometer en 2 meses por un ingeniero con experiencia a un ritmo de 8 horas diarias, distribuidos de forma equivalente, tanto en el diseño del Hardware como en el desarrollo de la librería, es decir, 1 mes para cada parte. El coste/hora de un ingeniero ronda los 35€/h, por tanto el coste humano para el desarrollo ascendería a la cifra de 11.200€ para 320 horas.

Cantidad	Componente	Precio
1	Placa de circuito	2,20€
1	Montaje	1,76€
1	MCP2515	1,51€
1	MCP2551	1,02€
1	Cristal 16MHz	0,52€
4	Bloque atornillable 3 vías	0,30€
4	Bloque atornillable 2 vías	0,25€
1	Pack pines shield	1,88€
1	Switch Reset	0,33€
1	Fusible reseteable 0,5A	0,37€
1	Led amarillo	0,05€
1	Led verde	0,05€
1	Led rojo	0,05€
2	Condensador 0,1μF	0,006€
2	Condensador 22ρF	0,01€
1	Condensador 10μF	0,07€
1	Resistencia 10KΩ	0,02€
3	Resistencia 1KΩ	0,02€
1	Resistencia 120Ω	0,02€
	Total	**12,10€**

Tabla 4.2: Tabla de costes para un Shield

Señalar que es un coste estimado y que generalmente, en proyectos de este tipo se trabaja a "Proyecto Cerrado" dando una cifra mucho más baja, incluso la mitad de la propuesta y olvidándonos del coste/hora, más aún, en la actual panorama de crisis que vivimos.

Capítulo 5

Software

5.1. Comunicación SPI

Configuración

Lo primero que debemos implementar es la comunicación más básica, la que se producirá entre Arduino y nuestro shield, para que en capas superiores de Software podamos enviar y recibir mensajes CAN. Esta comunicación se realiza sobre SPI y debemos prestar una especial atención porque serán los pilares de nuestro sistema.

Básicamente el SPI se implementa con un registro de desplazamiento síncrono, creo que la siguiente figura es muy instructiva y está sacada del datasheet del ATmega8.

En este caso, el dispositivo como Master es el encargado de generar la señal de reloj, si nos fijamos un poco, podemos ver que cuando se produzcan 8 pulsos en SCK, los datos contenidos en los registros se habrán intercambiado en su totalidad y de esta forma se comunican.

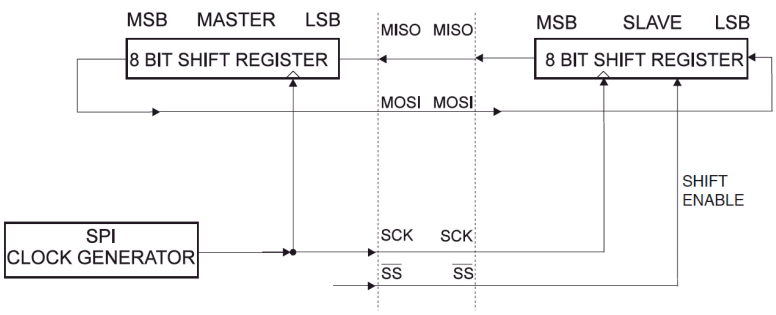


Figura 5.1: SPI Interconexión Maestro-Esclavo

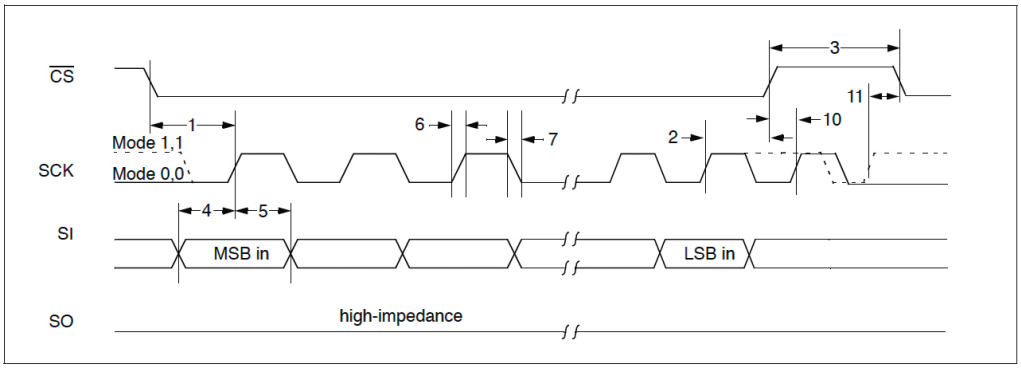


Figura 5.2: SPI Input Timing

En las especificaciones de SPI existen básicamente 4 modos de comunicación, en términos generales respetan la misma política pero sin embargo no son compatibles y dependerá del dispositivo que usemos. La diferencia entre ellos es el nivel de tensión para arrancar y los flancos de la señal de reloj en los que se toma la muestra. En nuestro caso particular y teniendo en cuenta que lo que queremos es comunicarnos con el CAN-Controller MCP2515, debemos acudir a las especificaciones técnicas dadas por el fabricante para saber en que modo debemos configurar Arduino.

Según el datasheet del MCP2515, es un dispositivo esclavo, en el capítulo 12, página 63, están definidas todas las especificaciones referentes al interfaz SPI. Por tanto, la configuración del periférico de comunicaciones en Arduino deberían ser MODO 0,0 (0) o el 1,1 (3).

Tal y como vemos en el cronograma, lo primero que debemos hacer es activar al dispositivo bajando la línea CS, ChipSelect, es necesario hacerlo de forma manual y explícita en nuestro programa, pues el periférico en el AT-Mega8 no tiene una línea ChipSelect "Automática", entre otras cosas porque no sería interesante puesto que anclaría el funcionamiento del bus a la comunicación con un único dispositivo, por lo que es más sencillo y polivalente, si podemos controlar las líneas de ChipSelect por separado en cada uno de los elementos conectados al bus SPI. Una vez activado, tendremos que mandar el dato por SI, y poner a funcionar el reloj SCK. Una vez terminada la transferencia de un Byte, ponemos a nivel alto CS, para deseleccionar el dispositivo y paramos nuestro reloj SCK.

En nuestro caso, AT-mega8, que es el microcontrolador que incorpora Arduino, cuenta con una Interfaz SPI Hard, esto facilita nuestro trabajo que simplemente se reduce a configurar, mediante los registros adecuados, este periférico. Los registros son:

- SPCR (SPI Control Register): Es el que nos permite configurar los distintos aspectos del periférico, interrupciones, habilitación, modo, Master/Slave y frecuencia para el SCK.
- SPSR (SPI Status Register): Este es el registro de estado y nos informa mediante los distintos Flag/bits el estado del periférico.
- SPDR (SPI Data Register): Este es el registro donde se vuelcan los Bytes a enviar y tras el envío encontraremos en él, el Byte recibido.

SPI™ Interface AC Characteristics			Industrial (I): TAMB = -40°C to +85°C VDD = 2.7V to 5.5V Extended (E): TAMB = -40°C to +125°C VDD = 4.5V to 5.5V			
Param. No.	Sym	Characteristic	Min	Max	Units	Conditions
	FCLK	Clock Frequency	—	10	MHz	
1	TCSS	$\overline{CS}$ Setup Time	50	—	ns	
2	TCSH	$\overline{CS}$ Hold Time	50	—	ns	
3	TCSD	$\overline{CS}$ Disable Time	50	—	ns	
4	TSU	Data Setup Time	10	—	ns	
5	THD	Data Hold Time	10	—	ns	
6	TR	CLK Rise Time	—	2	µs	Note 1
7	TF	CLK Fall Time	—	2	µs	Note 1
8	THI	Clock High Time	45	—	ns	
9	TLO	Clock Low Time	45	—	ns ns	
10	TCLD	Clock Delay Time	50	—	ns	
11	TCLE	Clock Enable Time	50	—	ns	
12	TV	Output Valid from Clock Low	—	45	ns	
13	THO	Output Hold Time	0	—	ns	
14	TDIS	Output Disable Time	—	100	ns	

Note 1: This parameter is not 100% tested.

Figura 5.3: Características del interfaz SPI

Además es necesario conocer a la velocidad del bus SPI de nuestro dispositivo CAN-Controller, para poder ajustar el SCK según las características,

El primer parámetro es el que buscamos, 10MHz, como Arduino trabaja con cristales de 16MHz, es necesario configurar el divisor en 2, y trabajar con un SCK de 8MHz. Para configurar la frecuencia de reloj es necesario escribir en los registros del AtMega8/16, concretamente son dos los que tendremos que configurar, el SPI Control Register (SPCR) y el SPI Status register (SPSR). Como hemos dicho, la F_{OSC} de Arduino es de 16MHz por tanto buscamos la configuración correcta para que $F_{SCK} = F_{OSC}/2$ haciendo uso de la Tabla 5.1.

SPI2X	SPR1	SPR0	F_{SCK}
0	0	0	$F_{OSC}/4$
0	0	1	$F_{OSC}/16$
0	1	0	$F_{OSC}/64$
0	1	1	$F_{OSC}/128$
1	0	0	$F_{OSC}/2$
1	0	1	$F_{OSC}/8$
1	1	0	$F_{OSC}/32$
1	1	1	$F_{OSC}/64$

Tabla 5.1: Tabla de configuración para el SCK

Instruction Name	Instruction Format	Description
RESET	1100 0000	Resets internal registers to default state, set Configuration mode.
READ	0000 0011	Read data from register beginning at selected address.
Read RX Buffer	1001 0nm0	When reading a receive buffer, reduces the overhead of a normal read command by placing the address pointer at one of four locations, as indicated by 'n,m'. **Note:** The associated RX flag bit (CANINTF.RXnIF) will be cleared after bringing $\overline{CS}$ high.
WRITE	0000 0010	Write data to register beginning at selected address.
Load TX Buffer	0100 0abc	When loading a transmit buffer, reduces the overhead of a normal Write command by placing the address pointer at one of six locations as indicated by 'a,b,c'.
RTS (Message Request-To-Send)	1000 0nnn	Instructs controller to begin message transmission sequence for any of the transmit buffers. 1000 0nnn — Request-to-send for TXB2 — Request-to-send for TXB1 — Request-to-send for TXBO
Read Status	1010 0000	Quick polling command that reads several status bits for transmit and receive functions.
RX Status	1011 0000	Quick polling command that indicates filter match and message type (standard, extended and/or remote) of received message.
Bit Modify	0000 0101	Allows the user to set or clear individual bits in a particular register. **Note:** Not all registers can be bit-modified with this command. Executing this command on registers that are not bit-modifiable will force the mask to FFh. See the register map in **Section 11.0 "Register Map"** for a list of the registers that apply.

Figura 5.4: Conjunto de Comandos SPI

Programación

Para realizar la comunicación con nuestro CAN-Controller es necesario conocer el juego de instrucciones que implementa. En el datasheet podemos encontrar la siguiente tabla de OpCodes.

Haremos una breve descripción de los comandos que tenemos disponibles,

- RESET: tan solo debemos enviar 0xC0. Resetea todos los registros, y coloca al MCP2515 en "Modo Configuración", en futuras entradas trataremos el "Modo de Configuración" más ampliamente.

- READ: para leer un registro basta con enviar 0×03 seguido de la dirección del registro a leer "0xaddress" y a continuación enviamos un dummy (basura) para así recoger en SPDR el valor del registro leído.

- READ RX BUFFER: Podríamos decir que se trata de un "acceso directo" a los registros de control de los buffers de recepción de mensajes CAN. Su uso puede resultar un poco extraño, tendríamos que enviar 0b10010nm0 donde 'n' y 'm' nos marcaría el buffer al que queremos acceder. Como en el comando READ, finalizamos enviando un dummy para poder recoger el valor del registro en SPDR. En la siguiente tabla vemos las combinaciones posibles:

n	m	Address Points to	Address
0	0	Receive Buffer 0, Start at RXB0SIDH	0x61
0	1	Receive Buffer 0, Start at RXB0D0	0x66
1	0	Receive Buffer 1, Start at RXB1SIDH	0x71
1	1	Receive Buffer 1, Start at RXB1D0	0x76

Figura 5.5: OpCodes Read

- WRITE: Ya lo hemos usado, pero no está de más verlo de nuevo. Enviamos 0×02 seguido de la dirección del registro que deseamos modificar "0xaddress" y terminando con el dato a escribir.

- LOAD TX BUFFER: Este comando es otro "acceso directo" a los registros que van a controlar los buffers de emisión de datos via CAN. En este caso debemos enviar 0b01000abc seguido del dato que deseamos cargar, 'a', 'b' y 'c' usa un mecanismo similar al de READ RX BUFFER, la tabla para acceder a los distintos BUFFER de transmisión es:

a	b	c	Address Points to	Addr
0	0	0	TX buffer 0, Start at TXB0SIDH	0x31
0	0	1	TX buffer 0, Start at TXB0D0	0x36
0	1	0	TX buffer 1, Start at TXB1SIDH	0x41
0	1	1	TX buffer 1, Start at TXB1D0	0x46
1	0	0	TX buffer 2, Start at TXB2SIDH	0x51
1	0	1	TX buffer 2, Start at TXB2D0	0x56

Figura 5.6: OpCode LoadTX

- REQUEST-TO-SEND(RTS): Básicamente es la que da la orden de empezar la transmisión de los datos que están almacenados en los buffers. Su funcionamiento es tan simple como enviar un solo comando, 0b10000nnn donde colocando un '1' en alguno de los n, daremos permiso a ese buffer para empezar la transmisión.

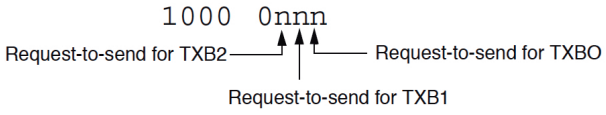


Figura 5.7: OpCode RTS

- READ STATUS: Este comando nos entrega el registro de estado de nuestro controlador CAN, se envia 0xA0, y enviamos un dummy, ahora podemos recoger el resultado de la lectura del SPDR, pero tenemos que tener en cuenta, que debemos enviar un dummy más, porque tal como está implementado, el comando envía por duplicado el registro de estado. En este caso puede que la forma más sencilla de tratar con este comando, sea enviando 0xA0 seguido de 2 dummys y posteriormente recoger en SPDR el registro de estado.

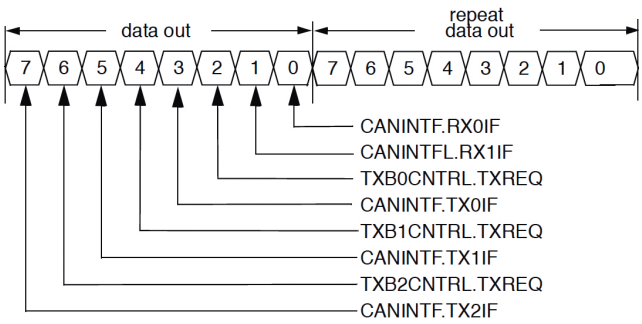
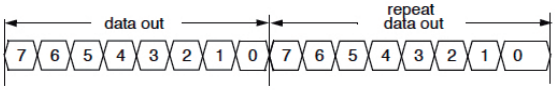


Figura 5.8: OpCode Status

- RX STATUS: Bastante similar al comando anterior, con la salvedad que en este caso, estamos interesados en consultar el estado de recepción. Se procede de igual forma pero en este caso el OpCode a enviar es 0xB0, y lo que recibimos es:



7	6	Received Message
0	0	No RX message
0	1	Message in RXB0
1	0	Message in RXB1
1	1	Messages in both buffers*

CANINTF.RXnIF bits are mapped to bits 7 and 6.

4	3	Msg Type Received
0	0	Standard data frame
0	1	Standard remote frame
1	0	Extended data frame
1	1	Extended remote frame

The extended ID bit is mapped to bit 4. The RTR bit is mapped to bit 3.

2	1	0	Filter Match
0	0	0	RXF0
0	0	1	RXF1
0	1	0	RXF2
0	1	1	RXF3
1	0	0	RXF4
1	0	1	RXF5
1	1	0	RXF0 (rollover to RXB1)
1	1	1	RXF1 (rollover to RXB1)

* Buffer 0 has higher priority, therefore, RXB0 status is reflected in bits 4:0.

Figura 5.9: OpCode RX Status

- BIT MODIFY: Este es el comando que más envío de datos requiere, se procede de la siguiente forma, enviamos 0×05 seguido de la dirección del registro que queremos modificar “0xaddress” y posteriormente enviamos la máscara de bits, que es donde codificaremos cuales son los bit que deseamos modificar, por ejemplo 0×81 (0b10000001) con esta máscara solo modificamos el primer y el último bit, a continuación enviamos el dato a modificar “0xdata”.

5.2. Modos de funcionamiento

El MCP2515 tiene 5 modos de funcionamiento que vamos a enumerar, explicar en que consisten y sus usos.

5.2.1. Configuration Mode

No hay que olvidar que antes de empezar a funcionar con nuestro CAN-Controller es necesario hacer una inicialización de este; para ello es necesario estar en este modo,

que es el único que nos permite modificar algunos de los registros necesarios para la correcta inicialización. Por defecto, el MCP2515 se encuentra en este estado después del power-up y un reset, además es posible cambiar en cualquier momento a este modo a través del registro CANTRL.REQOP. También hay que tener en cuenta, que en este modo se borran todos los marcadores de error.

5.2.2. Sleep Mode

Con este modo conseguimos bajar el consumo del dispositivo aunque el interfaz de comunicaciones SPI permanece activo pudiendo tener acceso a todos los registros. En caso de estar activa la interrupción, que se configura a través del registro CANINTE.WAKIE, mientras el MCP2515 está dormido podría detectar actividad en el bus y despertar al microprocesador, por tanto podemos tener al nodo dormido al completo y así conseguir un consumo muy reducido.

Una cosa importante a tener en cuenta, es que en el estado de dormido, el pin TXCAN permanecerá en estado recesivo y por tanto no responderá el ACK a los mensajes de la red, es decir, es como si no existiera el nodo.

El funcionamiento en modo dormido, junto con el wake-up es muy interesante aunque hay algunos detalles muy importante a destacar. En el modo dormido el dispositivo siempre monitoriza el pin RXCAN y en caso de que la interrupción CANINTE.WAKIE esté activada, generará el evento. En modo dormido el oscilador del MCP2515 está apagado y cuando detecta actividad en el bus, tiene un tiempo de encendido definido en $128T_{OSC}$, por tanto el dispositivo ignorará el mensaje que lo hizo despertar y cualquier otro que se produzca durante este tiempo. Cuando despierta, el dispositivo entra en el estado de "Listen Mode" y será la Microcontroller Unit (MCU) la encargarda de ponerlo en "Normal Mode" para establecer la comunicación si así es requerido.

Además, como extra, el dispositivo posee un filtro paso bajo configurable a través del registro CNF3.WAKFIL, con el que podremos evitar despertar todo el nodo por culpa de algún glitch.

5.2.3. Listen-only Mode

Este modo se puede usar como monitor del bus CAN ya que recibe todos los mensajes del bus, incluso los de error. Además, también es usado para detectar la velocidad o baud rate en conexión en caliente o hot plugging, para poder detectar la velocidad es necesario que al menos otros dos nodos de la red estén comunicándose.

En este modo no se podrán trasmitir mensajes, incluyendo en estos ACKs o tramas de error. El contador de error se resetea y se desactiva.

5.2.4. Loopback Mode

Este estado solo se usa en el desarrollo y testeo, lo que hace es conectar los buffers de transmisión con los de recepción, de forma que al enviar un mensaje este no saldrá al bus, sino que será recibido como un mensaje en el propio nodo. Por lo demás, su funcionamiento es similar al "Listen Mode", colocando el pin TXCAN en recesivo y no transmitirá ni tramas de error ni ACKs.

5.2.5. Normal Mode

Su modo de funcionamiento normal para operar en el bus. Se generan tramas de error, ACKs, etc. Es el único modo para transmitir mensajes al bus CAN.

5.3. Máscaras y filtros

Es necesario comprender que las comunicaciones a través de un bus CAN con un gran número de nodos puede implicar un tráfico de datos muy intenso, tan solo es necesario imaginar los sensores y actuadores que posee un automóvil para entender la cantidad de mensajes que podrían intercambiarse en un breve instante de tiempo. Si cada mensaje que navega por el bus debemos procesarlo con Arduino a nivel software para saber si es de nuestro interés, dedicaremos gran cantidad de recursos a este fin. Gracias a una de las características que nos aporta el CAN-Cotroller seleccionado, el MCP2515, podemos configurar unas máscaras y unos filtros de forma, que a nuestro Arduino tan solo lleguen los mensajes que nos interesan y procesando de esta forma lo necesario, sin desperdiciar los recursos del mismo.

El funcionamiento de los filtros y máscaras Hardware que nos pone a disposición el MCP2515 pueden parecer en un principio algo complicados de entender, por ello, veamos el diagrama de bloques junto a la tabla de verdad de la Figura 5.10 que encontraremos en el datasheet.

Como ya sabemos, el MCP2515 posee dos buffers de recepción etiquetados con los nombres RXB0 y RXB1. Según hemos visto en el diagrama de bloques de la Figura 5.10 cada uno de los buffers posee su propia cadena de filtrado. En el caso del RXB0 vemos como le corresponde la máscara 0 y los filtros 0 y 1, mientras que en el caso del RXB1, le corresponden la máscara 1 y los filtros 2,3,4 y 5.

Según lo que podemos extraer de la tabla de verdad de la Figura 5.10, la máscara indica los bits del identificador del mensaje que serán examinados por los filtros, por tanto, si quisiéramos examinar la totalidad del identificador estandard de un mensaje CAN, el valor de RXM0 debería ser 0x7FF, es decir los primeros 11 bits correspondiente al identificador a 1. En el caso, donde la máscara es 0x7FF, el identificador del mensaje debe coincidir en su totalidad con el valor que tengamos en alguno de los filtros, de no ser así no será aceptado.

Es interesante el uso de la máscara, configurada de forma correcta, nos permitiría

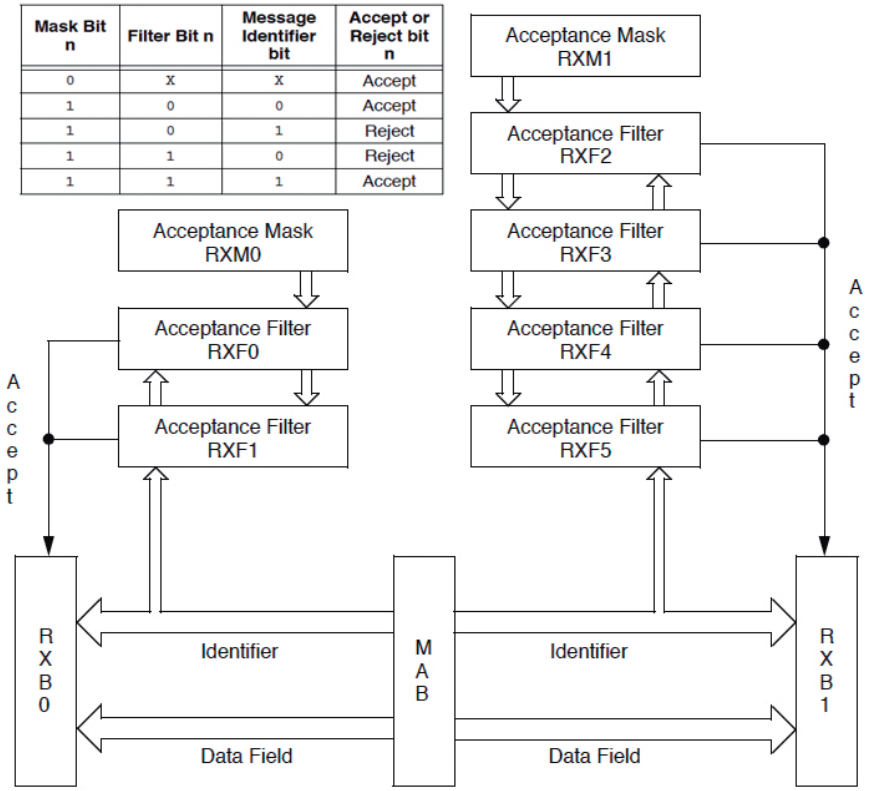


Figura 5.10: Máscara y Filtros CAN

aceptar un rango con distintos identificadores. Por ejemplo, si escribiésemos el valor 0x7F0 en RXM0 el valor del filtro con 0x2F0, los mensajes que se aceptarían tendrían el rango de 0x2F0 a 0x2FF, sin duda otra característica a tener en cuenta.

5.4. Librería para Arduino

Un buen punto de partida para conocer la forma correcta de programar librerías para Arduino es un documento, a modo de guía, que podemos encontrar en la web de Arduino, http://arduino.cc/es/Hacking/LibraryTutorial. Además de la forma con la que programar, explica los pasos necesarios para poder usar la librería en su entorno de desarrollo. La forma de hacer que el Integrated Development Environment (IDE) de Arduino use la librería, es colocar los ficheros fuentes en un árbol de directorios con una estructura concreta. La parte más tediosa, bajo mi punto de vista, es que cada vez que realicemos alguna modificación en la librería, tendremos que cerrar el programa de desarrollo, borrar el fichero compilado de la librería, cambiar los ficheros con el código fuente actualizado y volver a ejecutar el entorno de desarrollo. A mi parecer, es todo un atraso innecesario que a la larga complicará las cosas y las ralentizará.

Como alternativa, he decidido usar Eclipse como entorno de desarrollo y adaptarlo para que use Arduino y podamos tanto hacer los programas principales como el desarrollo de la librería sin tener tantas complicaciones. Hay que pensar, que Arduino, en lineas generales, está pensado para que lo usen personas con poca o

ninguna experiencia en el mundo de la ingeniería y la programación, por tanto, esa sencillez que lo ha hecho grande, es sin duda, una limitación cuando deseamos realizar proyectos más complejos.

Como hemos dicho, al trabajar con un entorno de desarrollo como Eclipse, tenemos una gran cantidad de ventajas, función auto-completar, multitud de plugin que facilitan la escritura de código e incluso podremos conectarlo con un servidor Sub-versión. En mi web, http://www.arcanplanet.es he publicado una guía de los pasos a seguir para configurar Eclipse con Arduino y poderlo usar para el desarrollo tanto de la librería como de programas completos.

5.4.1. Control de versiones

Cualquier software que se precie, debe usar un sistema de control de versiones. Aunque generalmente estos sistemas están pensados cuando sobre el mismo código trabajan varias personas, yo optaré por usarlo por dos motivos, el primero, nunca está demás aprender y el segundo, algún día el proyecto puede llegar a ser tan interesante que otros desarrolladores podrían querer participar en él.

Como Eclipse tiene un plugins que permite trabajar con Sub-versión, he elegido este sistema de control de versiones, que además, también es el que uso en la generación de la documentación de este proyecto. Otra de las ventajas de usar un control de versión, que hagamos lo que hagamos, siempre podremos volver atrás, al ser un proyecto con cierto volumen de trabajo, la seguridad que aporta un control versiones, sin duda, merece la pena.

5.4.2. Empezando con el código fuente

Con el fin de facilitar la programación he escrito un archivo de constantes, llamado *ArCan_def.h* donde se definen la correspondencia entre los pines que conectarán ArCan a Arduino. Además, en este mismo archivo, están definidas todos los registros y códigos de operación del $MCP2515$, así haremos mucho más legible el código fuente de la librería y podremos programar directamente usando el nombre de los registros tal y como vienen en el datasheet del dispositivo.

Sin duda alguna, como deseamos interactuar con el CAN-Controller, lo primero que deberemos hacer es crear los métodos que nos permita escribir y leer los registros del dispositivo de forma sencilla, serán métodos privados pues únicamente se usarán en las funciones y métodos de la propia librería.

5.4.3. Método: write_register

Es un método privado y por tanto solo es usable desde la propia librería. Se encarga de activar el dispositivo y escribir el registro que deseemos con el valor

indicado.

Definición:
void tArCan::write_register(byte address, byte data)

Sintaxis:
write_register(address, data);

Parámetros:
address Dirección del registro que queremos escribir.
data Valor con el que queremos configurarlo.

Return:
No devuelve nada.

Ejemplo:

```
//Configuración en modo normal.
write_register(CANCTRL, 0);
```

5.4.4. Método: read_register

Es un método privado y por tanto solo es usable desde la propia librería. Se encarga de activar el dispositivo y leer el registro que deseemos devolviendo el valor que tiene configurado.

Definición:
byte tArCan::read_register(byte address)

Sintaxis:
dato = ArCan.read_register(address);

Parámetros:
address Dirección del registro que queremos escribir.

Return:
Devuelve el valor que hay configurado en el registro que hemos especificado en address.

Ejemplo:

```
//Configuración en modo normal.
dato = read_register(CANCTRL);
```

Veamos los diferentes métodos y funciones que se han implementado como API para facilitar el control y la programación de ArCan. Como sabemos, vamos a implementar una librería de comunicaciones CAN para aislar al programador de más alto nivel del Hardware y proporcionarle un conjunto básico de funciones y métodos para controlar de forma sencilla las comunicaciones del bus. Cualquier librería de comunicaciones tiene un conjunto de funciones básicas, como podrían ser enviar o recibir datos, abrir la conexión, etc.

ArCan, siguiendo la línea de desarrollo de otros periféricos de comunicaciones en Arduino, se ha implementado como un objeto que posee una serie de métodos que permiten interactuar con él. Todo la funcionalidad es accesible mediante la clase *tArCan*, donde se definen todos los métodos a funciones básicos para el uso de ArCan, en ellos se implementa lo necesario para enviar/recibir cualquier mensaje "CAN 2.0 A".

Lo primero que encontramos en ArCan.h es la definición del registro $tCAN$ que contiene los parámetros de un mensaje "CAN 2.0 A", recordemos la Figura 3.4 donde podemos ver un Datagrama CAN.

```
typedef struct{
  int id;
  struct{
    byte rtr:1;
    byte length:4;
  }header;
  byte data[8];
}tCAN
```

Ahora veremos en detalle cada uno de los métodos implementados para ArCan y un ejemplo de funcionamiento para facilitar su uso y comprensión.

5.4.5. Método: begin

Al ejecutar este método se inicializa la shield ArCan con los siguientes pasos:

- Reset software a ArCan a través del comando SPI y así poner el dispositivo en un estado conocido.

- Configuración de los registros CNFx, que dependerán de la velocidad seleccionada.
- Testeo para verificar que ArCan es accesible.
- Configuración de los registros encargados del control de los Leds D2 y D3 para que se encienda en caso de que el buffer de recepción se encuentre lleno.
- Configuración de los pines TXnRTS como entradas digitales, para impedir que pudiesen estar en corto y provocar un sobreconsumo del integrado.
- Se desactivan todos los filtros en la recepción de mensajes.
- Cambiar al modo normal de funcionamiento.

Una vez terminada la secuencia de inicialización, nuestro dispositivo ArCan se encuentra en el estado correcto para operar con normalidad en el bus y notificar/recibir cualquier mensaje que circule por la red.

Definición:
boolean tArCan::begin(uint16_t speed)

Sintaxis:
ArCan.begint(velocidad);

Parámetros:
No requiere ningún parámetro.

Return:
True si toda ha sido correcto
False en caso contrario

Ejemplo:

```
if(!ArCan.begin(125))
  return 0;//
Código a ejecutar en caso de que la inicialización sea correcta
```

5.4.6. Método: check_message

Con este método podemos saber si existe algún mensaje en el buffer de recepción.

Definición:
boolean tArCan::check_message(void)

Sintaxis:
ArCan.check_message();

Parámetros:
No requiere ningún parámetro.

Return:
True Existe al menos un mensaje
False No hay ningún mensaje nuevo

Ejemplo:

```
if(acan.check_message())
  ArCan.get_message(&mensaje);
```

5.4.7. Método: check_free_buffer

Con este método podemos conocer si existe algún buffer de transmisión libre en ArCan.

Definición:
boolean tArCan::check_free_buffer(void)

Sintaxis:
check_free_buffer()

Parámetros:
No requiere ningún parámetro.

Return:
True Existe al menos un buffer de transmisión libre.
False En caso contrario, es decir, no hay disponible ningún buffer para realizar la transmisión.

Ejemplo:

```
if(ArCan.check_free_buffer())
  ArCan.send_message(&mensaje);
else
  print("Error al enviar: buffer llenos");
```

5.4.8. Método: get_message

Este método guarda los mensajes almacenados en el buffer de recepción del MCP2515, en una estructura de mensajes tCAN para el posterior procesamiento en Arduino.

Definición:
*byte tArCan::get_message(tCAN *msje)*

Sintaxis:
status_register=ArCan.get_message(&mensaje);

Parámetros:
Dirección de memoria a un registro del tipo tCAN.

Return:
Devuelve el registro de estado del MCP2515. Recordemos que el registro de estado del MCP2515 tiene la forma de la Figura 5.8.

Ejemplo:

```
num_mensajes=ArCan.check_message();
if(num_mensajes > 1)
  ArCan.get_message(&mensaje);
```

5.4.9. Método: send_message

Es el método que permite enviar mensajes CAN.

Definición:
*byte tArCan::send_message(tCAN *msje)*

Sintaxis:
status_register=ArCan.send_message(&mensaje);

Parámetros:
Dirección de memoria a un registro del tipo tCAN.

Return:
Devuelve la dirección del buffer por el que se enviará el mensaje CAN. En caso de estar los buffers llenos devolvería un 0.

Buffer	Address
TXBuffer 0	0x01
TXBuffer 1	0x02
TXBuffer 2	0x06

Tabla 5.2: Dirección de memoria de los buffers de transmisión

Ejemplo:

```
if(ArCan.check_free_buffer(){
  num_buffer=ArCan.send_message(&mensaje);
  if(num_buffer == 0)
    print("Error al enviar");
}else
  print("Error al enviar:Buffers llenos");
```

5.4.10. Método: SetMode

Función que cambia el modo de funcionamiento entre los posibles. Ver la Sección 5.2 para comprender el funcionamiento y la configuración de los modos de funcionamiento del MCP2515.

Definición:
void tArCan::SetMode(byte modo)

Sintaxis:
ArCan.SetMode(MODO);

Parámetros:
Se le pasa la constante de modo definida en el fichero ArCan_def.h:

- "CONFIGURATION": Activa el modo configuración.
- "NORMAL": Activa el modo normal.
- "SLEEP": Activa el modo dormido.
- "LISTEN": Activa el modo escucha.
- "LOOPBACK": Activa el modo para testing y desarrollo.

Return:
No devuelve nada.

Ejemplo:

```
// Configuramos los registros que deseemos.
ArCan.SetMode(CONFIGURATION);
// Activamos el modo normal.
ArCan.SetMode(NORMAL);
```

5.4.11. Método: SetRXB0Filters

Método que nos permite configurar la máscara y filtros asociados al buffer de recepción RXB0. Ver la Sección 5.3 para comprender el funcionamiento y la configuración de las máscaras y filtros.

Definición:
void tArCan::SetRXB0Filters(uint16_t Mask0, uint16_t filter0, uint16_t filter1)

Sintaxis:
ArCan.SetRXB0Filters(Mask0, Filter0, Filter1);

Parámetros:
Los valores a configurar para la máscara y a los dos filtros asociados al buffer RXB0.

Return:
No devuelve nada.

Ejemplo:

```
// Configuramos los filtros para que solo acepten
// los IDs: 0x7DD y 0x7DE
ArCan.SetRXB0Filters(0x7FF, 0x7DF, 0x7DF);
```

5.4.12. Método: SetRXB1Filters

Método que nos permite configurar la máscara y filtros asociados al buffer de recepción RXB1. Ver la Sección 5.3 para comprender el funcionamiento y la configuración de las máscaras y filtros.

Definición:
void tArCan::SetRXB1Filters(uint16_t Mask1, uint16_t filter2, uint16_t filter3, uint16_t filter4, uint16_t filter5)

Sintaxis:
ArCan.SetRXB0Filters(Mask1, Filter2, Filter3, Filter4, Filter5);

Parámetros:
Los valores a configurar para la máscara y a los dos filtros asociados al buffer RXB1.

Return:
No devuelve nada.

Ejemplo:

```
// Configuramos los filtros para que solo acepten
// los IDs: 0x7DA, 0x7DB, 0x7DD, 0x7DE
ArCan.SetRXB1Filters(0x7FF, 0x7DA, 0x7DB, 0x7DD, 0x7DE);
```

5.4.13. Soporte para interrupciones

En el inicio del desarrollo no se contemplaba el uso de una de las interrupciones externas que posee Arduino para que el MCP2515 pudiera interrumpir, en caso de que alguno de los buffers de recepción estuviera lleno, por tanto la forma de proceder era realizando un polling en el programa principal para testear el pin INT del CAN-Controller, configurado para activarse a nivel bajo en caso de que alguno de los dos buffers de recepción estuviera completo. Obviamente, esto causa cierta sobrecarga en Arduino y en las versiones posteriores de la librería se implementó el uso de la interrupción externa de Arduino para mejorar el rendimiento de nuestro nodo CAN.

El mayor problema que tendría realizar un polling, es que en programas con mayor complejidad y tiempo de procesado, se podrían perder mensajes si coincidiese que los dos buffers de recepción están llenos y Arduino no llega a realizar el polling y liberar los buffers antes de que se produjese una nueva comunicación en el bus. Para evitarlo, habría que prestar especial cuidado en la temporización con la que se realiza el polling, de forma, que sea menor que el tiempo de la transmisión mínima de un mensaje CAN.

El uso de interrupciones, aunque aumenta ligeramente la complejidad de la librería, sus ventajas bien merecen la pena. Podría ayudarnos a mejorar el consumo, puesto que podríamos tener en modo dormido a Arduino a la espera de mensajes, nos aseguramos de que no perdemos ningún mensaje y además evitamos la carga de procesado que implica el polling.

Para la implementación de esta mejora, se ha creado un buffer de recepción del tipo tCAN con 8 posiciones, es decir, capaz de almacenar hasta 8 mensajes CAN y un contador de la cantidad de mensajes que se encuentran almacenados en dicho buffer. También se ha añadido una función específica para almacenar los mensajes recibidos al nuevo buffer de recepción, que será la función que coloquemos en la rutina de interrupción.

Además de los cambios en el software, hay que hacer una conexión en Arduino para unir la salida del pin INT del MCP2515 a la entrada de interrupción externa. Teniendo en cuenta que la INT del CAN-Controller está rutada al pin Digital 9 de Arduino, los pines que hay que interconectar son el pin Digital 2(Interrupcion externa 0) con el pin 7(INT del MCP2515) tal y como vemos en la Figura 5.11.

5.4.13.1. Método: get_message_to_buffer_rx

Este método guarda los mensajes de los buffers de recpción del MCP2515 al buffer interno de la librería ArCan, para posteriormente ser procesados desde el programa principal de Arduino.

Definición:
byte tArCan::get_message_to_buffer_rx(void)

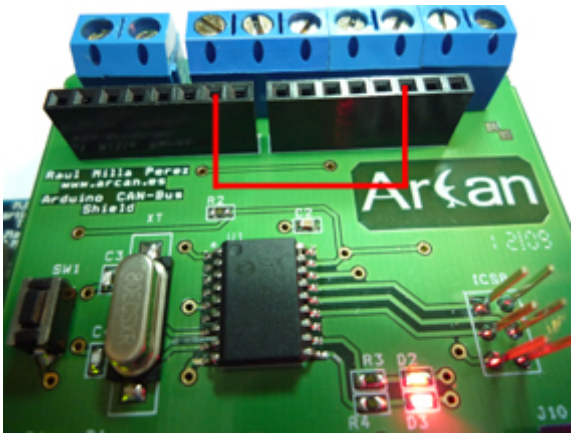


Figura 5.11: Habilitar Interrupción del MCP2515

Sintaxis:
status_register=ArCan.get_message_to_buffer_rx();

Parámetros:
No requiere ningún parámetro.

Return:
Devuelve el registro de estado del MCP2515. Recordemos que el registro de estado del MCP2515 tiene la forma de la Figura 5.8.

Ejemplo:
Dentro de la parte de setup del programa de Arduino añadiremos al final,

```
attachInterrupt(0, ArCanInterrupt, FALLING);
```

Después del loop, añadimos la rutina de interrupción.

```
void ArCanInterrupt()
{
  //Almacenamos el mensaje en el buffer_rx
  ArCan.get_message_to_buffer_rx();
}
```

Y en nuestro programa principal, para procesar los mensajes que se encuentran en el buffer de recepción de la librería usaremos un código similar a este:

```
if (ArCan.check_message())
  ArCan.get_message(&mensaje);
```

Capítulo 6

Pruebas y Resultados

Una vez que se ha concluido con el desarrollo del sistema, es importante someterlo a una serie de pruebas que determinen el correcto comportamiento del mismo. Nuestro proyecto es del tipo integración, por ello el plan que diseñaremos no contempla pruebas sobre el estandar ISO11898, pues es Microchip, el fabricante de nuestros circuitos integrados, los que certifican y se encargan que los dispositivos cumplen con la norma. Por ello, nosotros nos centraremos en verificar la correcta integración de nuestro circuito y su funcionamiento.

En nuestro caso tendremos dos planes de pruebas diferenciados, por un lado tenemos que realizar las pruebas físicas correspondientes al hardware que nos asegure su correcto ensamblaje, y por otro lado verificar que las comunicaciones y funcionalidad de la placa en conjunto con nuestra librería es correcta.

Por tanto, el objetivo que perseguimos es un pequeño plan de prueba que nos permita confirmar de forma relativamente sencilla el buen funcionamiento del circuito y así detectar posibles errores en la producción, para poder asegurar que las placas que se comercialicen están correctas y perfectamente funcionales.

En la Tabla 6.1 se describen los diferentes grupos de prueba que han sido propuestos para para formar parte de este plan. Cada grupo cuenta con una serie de casos de prueba, los cuales se definen en la Tabla 6.2.

Nuestro plan de prueba no va más allá porque en este proyecto no se ha implementado ninguna capa de aplicación, en tal caso, habría que diseñar uno más extenso,

Hardware	Garantizar que los niveles de tensión y corriente son correctos.
SPI	Garantizar la comunicación SPI entre el Arduino y el CAN-Controller.
CAN	Garantizar que las comunicaciones CAN son correctas.

Tabla 6.1: Grupos de Pruebas

que comprobase, que efectivamente, cumple con la norma concreta. Por ejemplo, si diseñáramos un nodo compatible con CANopen, entonces, deberíamos ampliar el plan y añadir las pruebas necesarias que certifiquen que nuestro nodo cumple con la norma.

Para realizar las distintas pruebas, se hará uso de diferentes medios, dependiendo del tipo de prueba. Estos medios se componen de un polímetro, con el que podamos medir tensión/corriente y de un Analizador lógico de señales que nos permita capturar los mensajes CAN enviados y recibidos. Para las pruebas de velocidad en la transmisión y también para comprobar que las señales CAN están limpias con relación al ruido, sería interesante contar con un osciloscopio que nos permita realizar medidas con mayor exactitud y directamente en el bus.

Además, dependiendo del grupo de pruebas que estemos realizando es necesario grabar en Arduino un programa concreto tal y como se especifica a continuación:

- **Hardware (HW):** Estas pruebas se realizarán con un Arduino que contenga el programa de pruebas "*Hardware.ino*".
- **SPI:** Estas pruebas se realizarán con un Arduino que contenga el programa de pruebas "*Harware.ino*".
- **CAN:** Estas pruebas se realizarán con un Arduino que contenga el programa de pruebas "*CAN.ino*".

Por supuesto y con el objetivo de que las pruebas presenten la mayor fiabilidad, siempre que sea posible usaremos un Arduino patrón y reservado a tal fin, de forma que podamos asegurar una base en la prueba correcta.

Los resultados de cada prueba serán incluidos en una hoja de toma de resultados tal y como vemos en la Tabla 6.3, en la que se refleja el resultado del plan de pruebas al que ha sido sometido el sistema. El resultado final será denotado como "PASA" o "FALLA", en caso de que alguna prueba falle, se hará un breve comentario describiendo el tipo de fallo.

El entorno de pruebas para CAN se compondrá de una red con dos nodos más el de prueba, para confirmar la interoperabilidad entre los nodos y porque a fin de cuentas, el objetivo del proyecto en la intercomunicación de Arduinos mediante CAN bus. Un buen banco de pruebas para verificar el funcionamiento de los nodos y las comunicaciones CAN, podría ser el panel de demostración de una aplicación domótica que encontraremos en el Apéndice C de este proyecto.

Pruebas de Hardware		
ID	**Nombre**	**Descripción**
HW-01	Inspección visual	Comprobar visualmente el aspecto de ArCan
HW-02	Led verde	Comprobar que el led verde de ArCan se ilumina al conectarlo
HW-03	Alimentación	Comprobar con el polímetro que la tensión en bornas del conector J8 es 5V
HW-04	Consumo	Comprobar con el polímetro que el consumo del conjunto ArCan+Arduino está entre 26mA y 29mA
Pruebas de comunicaciones SPI		
ID	**Nombre**	**Descripción**
SPI-01	Leds D2 y D3	Al conectar ArCan los leds D2 y D3 deben estar iluminados
SPI-02	Reset	Comprobar que al pulsar SW1 los leds D2 y D3 se apagan y encienden
SPI-03	Lectura de registro	Comprobar que el valor dado en la consola durante la inicialización es 0x0C
Pruebas de comunicaciones CAN		
ID	**Nombre**	**Descripción**
CAN-01	Mensaje Enviado	Comprobar que se envía un mensaje CAN correctamente, para ello usaremos el analizador lógico conectado al pin 1 del transceiver MCP2551, comprobaremos que el mensaje tiene el ID 0x7DF
CAN-02	Mensaje Recibido	Comprobar que se recibe un mensaje CAN correctamente, para ello usaremos el analizador lógico conectado al pin 4 del transceiver MCP2551, comprobaremos que el mensaje tiene el ID 0x7DF
CAN-01	250Kbps	El propósito de este caso de prueba es comprobar que la velocidad de transmisión es correcta.
CAN-02	500Kbps	El propósito de este caso de prueba es comprobar que la velocidad de transmisión es correcta.
CAN-03	1Mbps	El propósito de este caso de prueba es comprobar que la velocidad de transmisión es correcta.

Tabla 6.2: Plan de Pruebas

Pruebas de Hardware		
ID	**Resultado**	**Comentario**
HW-01		
HW-02		
HW-03		
HW-04		
Pruebas de comunicaciones SPI		
ID	**Resultado**	**Comentario**
SPI-01		
SPI-02		
SPI-03		
Pruebas de comunicaciones CAN		
ID	**Resultado**	**Comentario**
CAN-01		
CAN-02		
CAN-01		
CAN-02		
CAN-03		

Tabla 6.3: Hoja de resultados

Capítulo 7

Conclusiones y líneas futuras

En el inicio de este proyecto planteamos diversos objetivos que sin duda se han llevado a cabo, podemos decir que hemos creado con éxito un shield para Arduino que lo dota con la posibilidad de comunicarse con un bus CAN y desarrollado una librería que facilita su uso. Además de eso, hemos conseguido despertar cierto interés en la comunidad por el conocimiento que se ha ido plasmando en el blog del proyecto http://www.arcan.es a la vez que se realizó.

En mi opinión, el proyecto es muy completo, pues se han abarcado todas las fases para la creación de un producto real y funcional. Podríamos decir que consta de 4 grandes bloques, en los que hemos puesto gran esfuerzo para conseguir unos resultados con altos niveles de calidad.

- Estudio, análisis y definición: En esta primera fase se realizó del estado del arte con respecto a Arduino y a los buses CAN, donde se concluyó que era un proyecto innovador que cubría un mercado en expansión donde hemos sido pioneros, además de ser muy útil a fin de abrir las puertas a Arduino al mundo industrial. Con el paso del tiempo y una vez terminado la parte práctica del proyecto, se han observado como han ido saliendo productos similares a ArCan, pero puedo estar orgulloso, ArCan ha sido el primer shield para CAN bus de Arduino al igual que su librería.

- Desarrollo y diseño: Sin duda una de las partes más complejas del proyecto, aunque en la carrera se obtiene mucho conocimiento, enfrentarse a problemas reales es muy distinto. Esto nos ha dado una visión crítica y mucha perspectiva en líneas generales en lo que a Hardware se refiere y de forma particular a los componentes y tecnologías montadas en ArCan.

- Prototipos y producción: Empecé por realizar el montaje en un protoboard y comprobé el funcionamiento del primer diseño, al ser un circuito pequeño no fue muy complicado su implementación, después realicé el circuito en casa, usando placas fotosensibles, una insoladora que me construí con una caja de zapatos y una dremel, en este paso si que tuve varios problemas, tiempos de insolación, alguna pista cortada, el primer fotolito estaba impreso en modo espejo.... pero

a la segunda conseguí que funcionará, eso sí, modificando algunos problemas de conexionados en la PCB. Luego modifiqué el esquemático y la PCB para añadir los cambios y me atreví a enviar los gerbers, ficheros de fabricación, a fábrica del primer diseño funcional y probado. Posteriormente, y una vez trabajado con el hardware, encontré algunos problemas y posibles mejoras que añadí para las futuras producciones.

- Software y pruebas: La librería la fui desarrollando en paralelo con la evolución del hardware, este apartado es el que más tiempo ha llevado, es un software con cierta complejidad por ser de muy bajo nivel, así que la forma de proceder fue implementando las funciones más básicas que tocaban directamente los registros de Arduino y de ArCan para ir aumentando la complejidad y subiendo niveles de abstracción que facilitaran el trabajo y la posterior integración de ArCan a los usuarios finales.

Los objetivos iniciales del proyecto consistían únicamente en crear el Shield para Arduino y una librería para facilitar su uso a la comunidad. Sin embargo, una vez conocida la ventaja que aporta CAN como protocolo de comunicaciones, los objetivos se han visto claramente incrementados. Introducir el Hardware libre en general y a Arduino en particular en el mundo industrial o acercar las ventajas de CAN a la comunidad son alguna de las nuevas prioridades.

Otros de los posibles usos de Arduino gracias a ArCan es el mercado de la Automoción, con esta sinergia es posible crear un Sistema libre para diagnosis del automóvil, tan caros en la actualidad, rebajando drásticamente su coste final. Es evidente que ArCan abre las puertas del Hardware libre al mundo industrial, beneficiando sin duda a gran cantidad de empresas que encuentran trabas económicas a la hora de adquirir equipos CAN.

El conocimiento y experiencia práctica que me ha portado este proyecto ha sido clave en mi éxito laboral, obteniendo ofertas de trabajo de calidad en el sector de la ingeniería electrónica, el diseño de circuitos y la programación de software embebido. Además del reconocimiento por parte de la comunidad que se ha puesto en contacto conmigo en muchas ocasiones para agradecer que volcase parte del conocimiento adquirido a través del blog. Debido a esto, sería interesante que el proyecto continuara aportando nuevas mejoras al Hardware y a la libraría, al igual que añadiendo nuevas aplicaciones prácticas a modo de guías que puedan ser usados por lo usuarios a fin de poder despejar dudas sobre las tan interesantes redes CAN.

Con respecto a los planes de futuro a nivel Hardware, en base a la práctica y al aumento de conocimientos experimentados durante el desarrollo, veo dos líneas de actuación que deben ser paralelas y que conseguirían de ArCan un producto más competitivo.

- Interrupciones: Tal y como se planteó en un principio, el hardware no está preparado para soportar interrupciones y creo que sería un importante cambio que mejoraría las prestaciones del dispositivo. He estudiado las opciones, y el uso del pin INT que habilita el MCP2515. Actualmente este pin está rutado en la PCB a un pin digital estandard de Arduino. El caso es que Arduino solo soporta dos interrupciones externas que están en los pines digitales 2 y 3. Una

posible forma de actuación sería añadir una resistencia de 0Ω o un jumper que nos permita seleccionar ArCan con soporte para interrupción o en su forma original y poder mantener así la compatibilidad con las versiones anteriores.

- Conectores: Los conectores atornillables me siguen pareciendo una solución muy buena puesto que el uso de ArCan es de propósito general y no especifico para una aplicación concreta, donde el conector del bus podría estar definido. Aun siendo de este modo, me parece interesante añadir doble implantación para dicho conector, pudiendo ensamblar en caso de estimarlo oportuno un DB9 en vez del atornillable de 3 bornas, ya que según he podido ver, el DB9 es la opción más común.

Por otro lado, encontramos posibles mejoras en la Librería, esta ha ido sufriendo muchas modificaciones con el paso del tiempo y se han ido implementando bastantes mejoras, pero es algo que siempre tiene opciones de ampliación. Una de ellas fue con el fin de dar soporte a las interrupciones, para ello se creó un buffer donde almacenar los mensajes y un contador que nos indicaría la cantidad de mensajes almacenados, sin duda fue una mejora importante. Queda pendiente añadir soporte para "CAN 2.0B" y también para mensajes Multi-Frames, ampliaciones que me gustaría implementar en el futuro.

Apéndices

Apéndice A

ArCan

En la Tabla A.1 podemos encontrar la lista de componentes electrónicos que se montan en la placa de circuito. En la lista podemos encontrar la cantidad, la referencia al esquemático, el componente que se trata y por último, la referencia a DigiKey que he adquirido para el montaje de prototipos.

Existe la opción de montar o no el Choke ZJYA51R5-2PT. En caso de que nuestro entorno pueda presentar una tensión en modo común alta, estaría bien montarlo, su precio es aproximadamente de 1€. Si nos fijamos en la PCB, ha sido diseñada de forma que las resistencias R6 y R7 quedan por debajo de la bobina de choque, es por ello que ambos componentes nunca deben estar presentes. Si decidimos hacer el montaje estandard, R6 y R7 se montarán mientras que L1 no. Si decidimos incrementar el rechazo en modo común, entonces elegiremos montar L1 y dejar fuera R6 y R7.

Como vemos en el esquemático presentado en la Figura A.1, se ha realizado de forma modular para que podemos identificar la función de cada grupo de componentes, lo que sin duda facilita entender el funcionamiento.

En la Figura A.2 encontraremos impreso los gerbers del circuito, necesarios para su fabricación. En la Tabla A.2, vemos el nombre de las capas y su descripción. Además, en caso de externalizar el montaje, será necesario incluir las capas SPT y SPB, con ellas lo que se fabrica es el stencil o máscara para aplicar la pasta de soldadura antes de pasar la placa por la máquina de pick and place, la que monta los componentes de forma automática. Junto a los ficheros de las capas, será necesario entregar el fichero de aperturas "thruhole.tap", que es usado por la máquina de control numérico para realizar los taladros, también de forma automática.

Toda la información necesaria para la fabricación al igual que los gerbers se pueden encontrar en el CD adjunto al proyecto.

Q	REF	PART	DigiKey
1	CUARZO1	16MHZ 22ρF SMD	XC694CT-ND
1	C1	10μF 16V ELECT SMD	493-2100-1-ND
2	C2, C5	0,1μF 10V 0603 SMD	399-1095-1-ND
2	C3, C4	22ρF 10V 0603 SMD	399-1053-1-ND
1	D1	LED GREEN 0603 SMD	160-1183-1-ND
1	D2	LED RED 0603 SMD	160-1181-1-ND
1	D3	LED AMBER 0603 SMD	160-1180-1-ND
1	F1	POLYSWITCH 0,5A SMD	MINISMDC050FCT-ND
1	ICSP1	PIN HEADER STACK 4x2	Pack NKCElectronic
2	J1, J3	PIN HEADER STACK8x1	Pack NKCElectronic
2	J2, J6	PIN HEADER STACK 6x1	Pack NKCElectronic
1	J7	PIN HEADER 2x1	3M9447-ND
1	L1	ZJYA51R5-2PT	445-2219-1-ND
3	R1, R3, R4	1KΩ 0603 SMD	311-1.0KGRCT-ND
1	R2	10KΩ 0603 SMD	RHM10KGCT-ND
3	R5, R6, R7	0Ω 0603 SMD	P0.0GCT-ND
1	R8	120Ω 1206 SMD	311-120FRCT-ND
1	SW1	SWITCH TACT H=4.3MM	CKN9097-ND
1	U1	MCP2515 18SOIC SMD	MCP2515-I/SO-ND
1	U2	MCP2551 8SOIC SMD	MCP2551-I/SN-ND

Tabla A.1: Lista de componentes

Capa	Descripción
TOP	Capa de cobre superior, con el que hacer el fotolito
BOT	Capa de cobre inferior, con el que hacer el fotolito
SST	Capa de serigrafía superior
DRD	Capa de taladros, necesario para que la CNC sepa la herramienta a usar
SMT	Máscara de soldadura superior, se emplea al barnizar la PCB
SMB	Máscara de soldadura inferior, se emplea al barnizar la PCB
SPT	Máscara para pasta soldadura superior, se emplea para aplicar la pasta de soldadura para los componentes SMD

Tabla A.2: Nomenclatura en los Gerbers

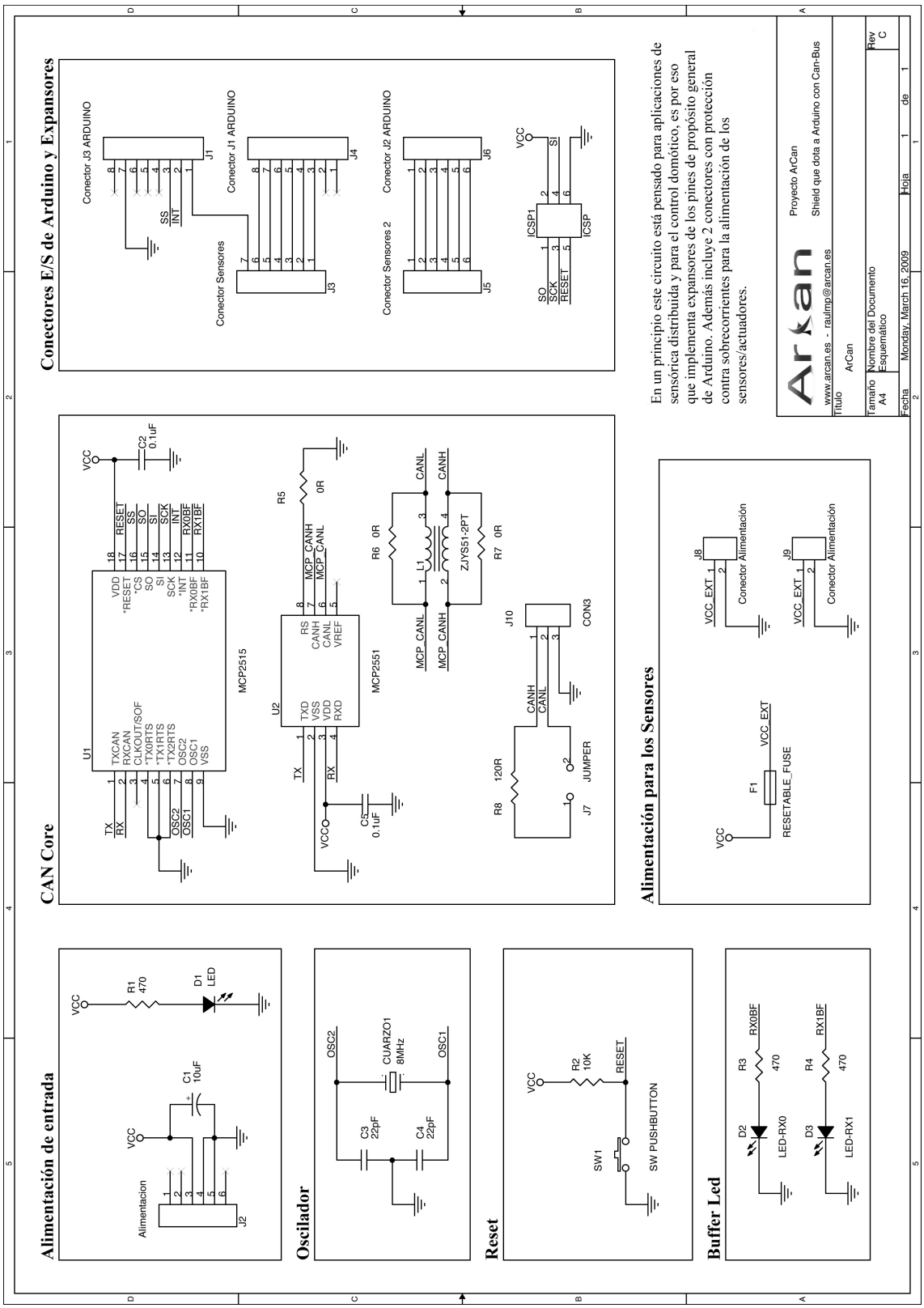
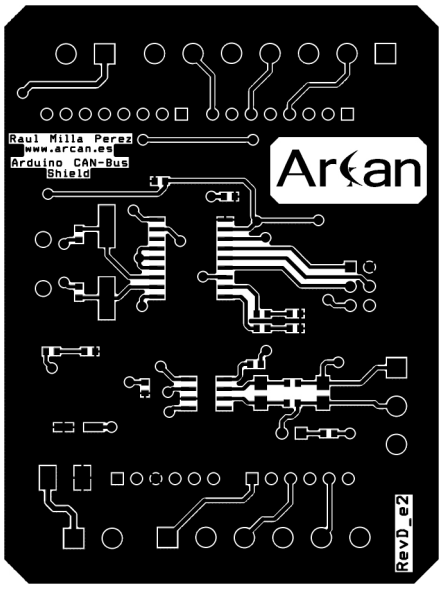
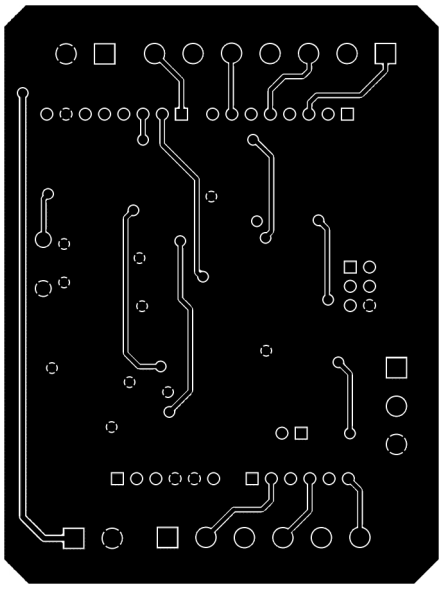


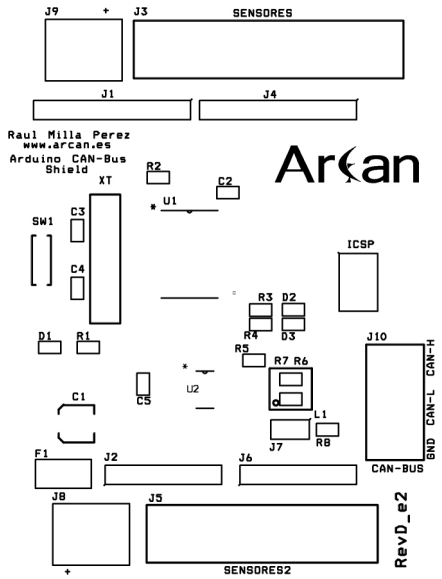
Figura A.1: Esquemático de ArCan Rev C



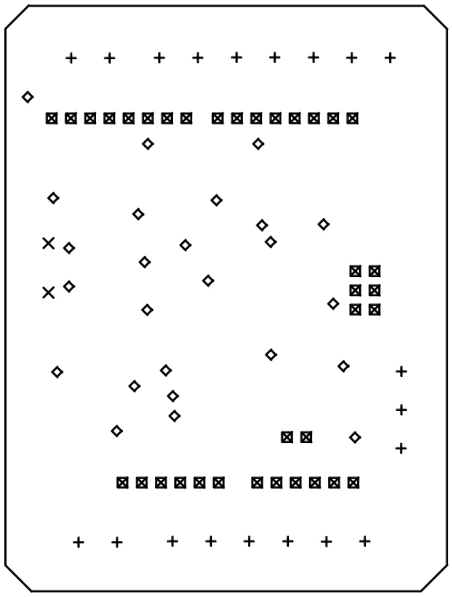
(a) Capa TOP



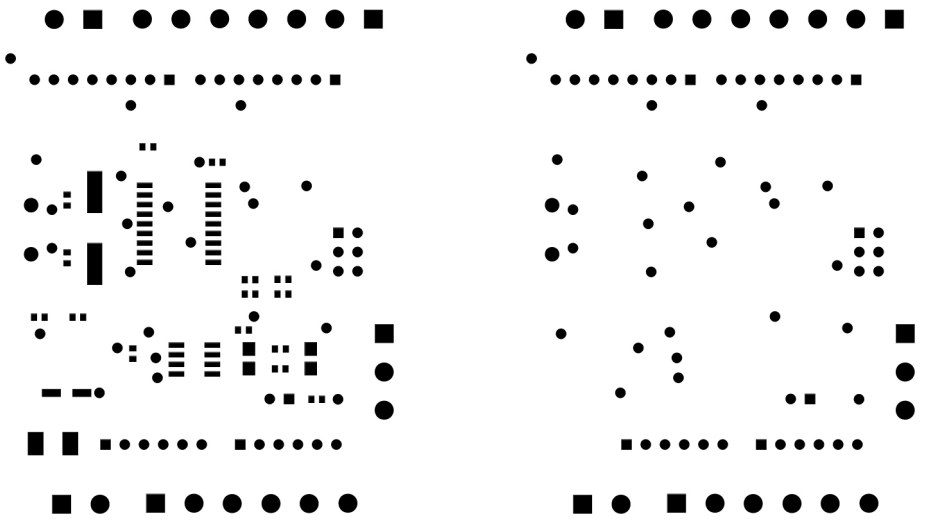
(b) Capa BOT



(c) Capa SST



(d) Capa DRD



(e) Capa SMT (f) Capa SMB

Figura A.2: Gerbers

Apéndice B

Documentos del proyecto

Se han creado 3 documentos para mejorar el impacto del proyecto en la comunidad y en los posibles clientes consumidores de este tipo de dispositivo. De estos proyectos, dos son guías o manuales destinados a facilitar el primer contacto y el tercero está más enfocado al marketing y venta del producto.

B.1. Manual de Hardware

Se ha creado un pequeño manual sobre el Hardware usado y como identificar sus componentes, el objetivo es facilitar su uso en la medida de lo posible, a las personas que puedan estar interesadas en este producto. Es un resumen del conocimiento expuesto a lo largo del proyecto, sintetizando lo más importante y necesario con el fin de poder usar ArCan.

Este manual se puede encontrar en el CD adjunto al proyecto o en la página web `http://www.arcan.es`***.***

B.2. Manual del Software

Se trata del último documento preparado, es una guía de referencia a la librería con pequeños ejemplos de uso. Es un magnífico punto de partida para las personas que se están iniciando en el mundo del CAN bus usando ArCan y Arduino como Hardware. Sin duda la síntesis de la información más importante y expuesta de la forma más sencilla posible.

Este manual se puede encontrar en el CD adjunto al proyecto o en la página web `http://www.arcan.es`***.***

B.3. Flyer publicitario

Este documento tiene el objetivo de ser un breve resumen de la importancia que puede tener este proyecto y persigue impactar a la persona interesada para que pueda entender y descubrir en un primer vistazo las cualidades más llamativas y las ventajas más significativas de lo que ArCan supone para Arduino.

Este flyer publicitario se puede encontrar en el CD adjunto al proyecto o en la página web `http://www.arcan.es`***.***

Apéndice C

Aplicación Domótica

Tal y como menciona el título de este proyecto, “Intercomunicación de Arduinos mediante Can-Bus”, hemos preparado un pequeño entorno de pruebas donde simular una posible aplicación de varios nodos CAN, compuesto cada uno por un Arduino y un Shield ArCan. En esta aplicación, haremos uso tanto del hardware diseñado en el proyecto como de la librería desarrollada para tal fin. Para entender el planteamiento principal podemos observar en la Figura C.1 un diagrama de la aplicación que intentaremos plasmar.

Para el entorno de pruebas, se ha preparado un pequeña maqueta con el plano de una casa, como vemos en la Figura C.2 que constará de algunos leds que simulan la luminaria de la casa, un zumbador, para simular una alarma y dos pulsadores para armar y activar dicha alarma. La maqueta, está compuesta por dos Arduinos, que usan sus pines digitales conectados a los distintos elementos de la maqueta. Además, hará falta un tercer Arduino, conectado a un Personal Computer (PC) con un software diseñado para tener el control de la maqueta.

El software de PC se ha realizado usando LabwindowsCVI, en la pantalla de inicio de la aplicación, podemos ver el mismo plano que encontramos en la maqueta y que vemos en la Figura C.2. Sobre el plano hay programados varios controles que coinciden en ubicación y función con la maqueta. A través del plano expuesto en la pantalla de la aplicación diseñada con LabWindows podremos actuar de forma

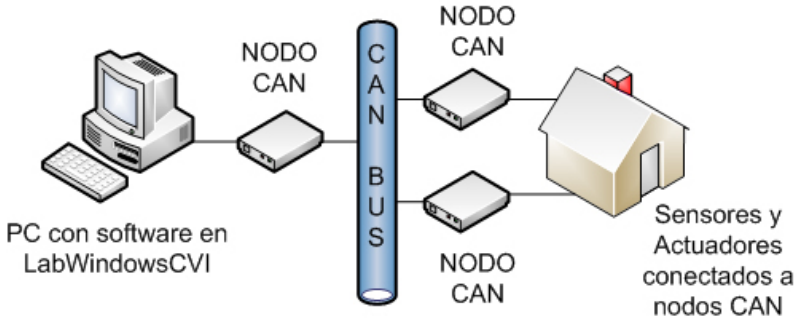


Figura C.1: Diagrama del panel de pruebas

sencilla e intuitiva sobre el panel. Para la correcta conexión de la maqueta y el software de acPC, será necesario configurar antes de cualquier operación el puerto serie de nuestro acPC donde tendremos conectado el nodo que hará de pasarela con los nodos de la maqueta.

A la hora de construir el panel y el entorno de pruebas se ha pensado en que la comunicación debía ser bidireccional, es por ello, que en el panel encontraremos dos tipos de elementos, elementos indicadores, como pueden ser los leds o el zumbador y elementos de control, como son los pulsadores para armar y activar la alarma. Veamos los casos que se van a producir y como actuaría cada nodo.

- El PC activa un indicador: En este caso, desde el panel se activaría un indicador, esto produciría un mensaje hacia el Nodo conectado al PC y que este codificaría y enviaría a través de la red CAN. Este mensaje sería recibido por ambos nodos, dependiendo que indicador queramos activar lo procesará un nodo u otro.
- Desde el panel se arma la alarma: El nodo que tiene conectado el pulsador de "Armar Alarma" detecta la pulsación y envía un mensaje CAN. Este mensaje llega al PC y este ilumina el indicador de "Alarma Armada".
- Desde el panel se activa la alarma: En este caso, el nodo que tiene conectado el pulsador de "Armar Alarma" detecta la pulsación y envía el mensaje. Este mensaje se recibe en el PC y en el otro nodo. Al recibir el mensaje de activación de alarma, entonces, las luces se ponen a parpadear, es decir, el nodo que se encarga de las luces ve que ese mensaje le interesa y lo procesa. A la vez, el panel cambia el indicador y avisa de que está sonando la alarma. También el propio nodo que hace saltar la alarma activa la señal acústica a través del zumbador. Esta situación es interesante, porque el mismo mensaje produce distintos comportamientos según el nodo.
- Desde el panel se desactiva la alarma: Se envía un mensaje CAN diciendo que la alarma está desactivada. Uno de los nodos apagará todas las luces y se pondrá en reposo, el otro nodo, parará el zumbador y pondrá los pulsadores en reposo. El propio panel también se pone en reposo apagando todos los indicadores. Como en el caso anterior, el mismo mensaje CAN es usado por varios nodos y cada uno desarrolla una acción distinta.

Cuando activemos algún control en el panel, se enviará un comando al Arduino conectado al PC y este, a través de ArCan enviará un mensaje CAN al panel, de forma, que cada nodo, sabrá que hacer en función del mensaje recibido.



Figura C.2: Panel de pruebas

Bibliografía

[1] Anonimous. http://rs232-rs485.blogspot.com.es/search/label/CAN.

[2] Steve Corrigan. Controller area network physical layer requirements, January 2008.

[3] Steve Corrigan. Common-mode effects in can networks, January 2012.

[4] Colaboradores de Wikipedia. Can bus. http://en.wikipedia.org/wiki/CAN_bus.

[5] International Organization for Standardization. Iso11898-1, road vehicles - controller area network - part 1 data link layer and physical signalling, Dicember 2003.

[6] International Organization for Standardization. Iso11898-2, road vehicles - controller area network - part 2 high speed medium access unit, Dicember 2003.

[7] Luca Fanucci Gianluca Casarosa, Michele Apuzzo and Bruno Sarti. Characterization of the emc performances of the can bus in a typical system bus architecture for small satellites, 2006.

[8] Robert Bosch GmbH. Can specification, 1991.

[9] CAN in Automation. http://www.can-cia.org.

[10] J. Lepkowski and B. Wolfe. Emi/esd protection solutions for the can bus, 2005.

[11] Jim Lepkowsky and Paul Lem. Tvs diode selection guidelines for the can bus, August 2004.

[12] Microchip. High-speed can transceiver, 2003.

[13] Microchip. Stand-alone can controller with spi interface, April 2005.

[14] Olaf Pfeiffer. Selecting a can controller. http://www.esacademy.com/en/library/technical-articles-and-documents/can-and-canopen/selecting-a-can-controller.html.

[15] Pat Richards. A can physical layer discussion, 2002.

[16] Ole-Kristian Skroppa and Scott Monroe. Common mode chokes in can networks: Source of unexpected transients, January 2008.

[17] Arduino Team. Arduino. http://arduino.cc.

[18] Wilfred Voss. *A Comprehensible Guide to Controller Area Network*. Copperhill Media, 2005.

[19] Conal Watterson. Controller area network (can) implementation guide, February 2012.

[20] Axel Wolf. Can - controller area network tutorial, August 1999.